NPSNET: A GRAPHICAL BASED EXPERT SYSTEM TO MODEL P-3 AIRCRAFT INTERACTION WITH SUBMARINES AND SHIPS

by
Dennis Arvin Schmidt
Commander, United States Navy
B.S.E.E., University of Texas, 1975

Submitted in partial fulfillment of the
requirements for the degree of

MASTER OF COMPUTER SCIENCE

from the

NAVAL POSTGRADUATE SCHOOL
June 1993

RITY CLASSIFICATION OF THIS PAGE

REPORT DOCUMENTATION PAGE

EPORT SECURITY CLASSIFICATION UNCLASSIFIED	1b. RESTRICTIVE MARKINGS
ECURITY CLASSIFICATION AUTHORITY	3. DISTRIBUTION/AVAILABILITY OF REPORT Approved for public release; distribution is unlimited
ECLASSIFICATION/DOWNGRADING SCHEDULE	
ERFORMING ORGANIZATION REPORT NUMBER(S)	5. MONITORING ORGANIZATION REPORT NUMBER(S)

AME OF PERFORMING ORGANIZATION npuţer Science Dept. al Postgraduate School	6b. OFFICE SYMBOL *(if applicable)* CS	7a. NAME OF MONITORING ORGANIZATION Naval Postgraduate School
DDRESS *(City, State, and ZIP Code)* nterey, CA 93943-5000		7b. ADDRESS *(City, State, and ZIP Code)* Monterey, CA 93943-5000
AME OF FUNDING/SPONSORING RGANIZATION	8b. OFFICE SYMBOL *(if applicable)*	9. PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER

DDRESS *(City, State, and ZIP Code)*

10. SOURCE OF FUNDING NUMBERS

PROGRAM ELEMENT NO.	PROJECT NO.	TASK NO.	WORK UNIT ACCESSION

ITLE *(Include Security Classification)*
NET: A GRAPHICAL BASED EXPERT SYSTEM TO MODEL P-3 AIRCRAFT INTERACTION WITH SUBMARINES AND SHIPS (U)

ERSONAL AUTHOR(S)
midt, Dennis Arvin

TYPE OF REPORT ster's Thesis	13b. TIME COVERED FROM 07/91 TO 06/93	14. DATE OF REPORT *(Year, Month, Day)* June 1993	15. PAGE COUNT 89

UPPLEMENTARY NOTATION The views expressed in this thesis are those of the author and do not reflect the official cy or position of the Department of Defense or the United States Government.

COSATI CODES			18. SUBJECT TERMS *(Continue on reverse if necessary and identify by block number)*
ELD	GROUP	SUB-GROUP	Graphics, P-3, Expert Systems, Torpedo Ballistics, Sonobuoy Ballistics, CLIPS

BSTRACT *(Continue on reverse if necessary and identify by block number)*

Computer Science Department at Naval Postgraduate School in Monterey, California has developed a low-co lespace simulation system, known as NPSNET, to work on commercially available Silicon Graphics IRIS wor ons. Initial work on NPSNET has concentrated primarily on ground-based forces with only limited work focusi naval or maritime air forces. With the present movement of the military towards totally integrated joint force o ions, there exists a need to expand existing modeling and simulation programs to include all aspects of militar rations. This thesis takes a step in that direction by incorporating naval maritime air units into NPSNET, expar its capability to include naval and Antisubmarine Warfare (ASW) units. This work focuses on several areas of arch, including modeling of the P-3 aircraft, aircraft motion control, aircraft ordnance ballistics modeling, inte on networking using the Distributed Interactive Simulation (DIS) protocol and development of an expert syste utonomously control aircraft behavior.

ISTRIBUTION/AVAILABILITY OF ABSTRACT UNCLASSIFIED/UNLIMITED ☐ SAME AS RPT. ☐ DTIC USERS	21. ABSTRACT SECURITY CLASSIFICATION UNCLASSIFIED	
NAME OF RESPONSIBLE INDIVIDUAL Michael J. Zyda	22b. TELEPHONE *(Include Area Code)* (408) 656-2305	22c. OFFICE SYMBOL CS/ZK

ORM 1473, 84 MAR — 83 APR edition may be used until exhausted. All other editions are obsolete

SECURITY CLASSIFICATION OF THIS PAGE
UNCLASSIFIED

ABSTRACT

The Computer Science Department at Naval Postgraduate School in Monterey, California has developed a low-cost battlespace simulation system, known as NPSNET, to work on commercially available Silicon Graphics IRIS workstations. Initial work on NPSNET has concentrated primarily on ground-based forces with only limited work focusing on naval or maritime air forces. With the present movement of the military towards totally integrated joint force operations, there exists a need to expand existing modeling and simulation programs to include all aspects of military operations. This thesis takes a step in that direction by incorporating naval maritime air units into NPSNET, expanding its capability to include naval and Antisubmarine Warfare (ASW) units. This work focuses on several areas of research, including modeling of the P-3 aircraft, aircraft motion control, aircraft ordnance ballistics modeling, interstation networking using the Distributed Interactive Simulation (DIS) protocol and development of an expert system to autonomously control aircraft behavior.

C.1

TABLE OF CONTENTS

LIST OF FIGURES

ACKNOWLEDGEMENTS

This project is not the work of one individual alone. There is a large number of people who contributed to this work both directly and indirectly. Although it would be impossible to acknowledge all of them, I would like to take this opportunity to thank some of the major players. Dr. Michael Zyda and Mr. David Pratt provided endless support and encouragement during this work. Dave never once hesitated to provide technical guidance regardless of the late night hour. Their guidance was invaluable.

I would also like to thank all of my classmates in the CS21 Graphics Track for their constant willingness to share ideas and provide recommendations for improvement. Dave Young helped me see the light in numerous programming dark holes and John Hearne made the work enjoyable with his never ending sense of humor.

Finally, I would like to thank my wife, Marsha, and my daughters, Angie and Shelly, for their loving support and understanding during the many nights I spent at the terminal in the Graphics Lab.

Without the help and moral support of the above people and countless others, this work would not have been possible.

I. INTRODUCTION

A. STATEMENT OF PROBLEM

Even in the current environment of shrinking budgets and downsizing of military forces, it is essential that our military units remain in the highest possible states of combat readiness. Rising fuels costs and the prohibitive costs of developing new combat systems have made it more difficult to train military forces effectively in full scale combat exercises than ever before. Current state of the art interactive simulation systems have leaped to the forefront as an effective yet economical alternative to full scale exercises. While sitting in an interactive simulator, the modern warrior is able to conduct repeated battle exercises with virtually an unlimited number of other players. The operator can safely train in a realistic environment at a small fraction of the cost of fueling a squadron of aircraft or a fleet of ships.

The Computer Science Department at Naval Postgraduate School in Monterey, California has developed a low-cost battlespace simulation system, known as NPSNET [Zyda92]. NPSNET is designed to work on off-the-shelf Silicon Graphics IRIS workstations. Initial work on NPSNET has concentrated primarily on ground-based forces with only limited work focusing on naval or maritime air forces. With the present movement of the military towards totally integrated joint force operations, there is a significant need to expand existing modeling and simulation programs to include all aspects of military operations. This thesis takes a step in that direction by incorporating naval maritime air units into NPSNET, expanding its capability to include naval and Antisubmarine Warfare (ASW) units.

B. FOCUS

The primary purpose of this research is to develop a proof of concept model for the P-3 aircraft working with a submarine in an ASW environment in NPSNET. This work focuses on several areas of research, including graphically modeling the P-3 aircraft,

aircraft motion control, aircraft ordnance ballistics modeling, interstation networking using the Distributed Interactive Simulation (DIS) protocol and development of an expert system to autonomously control aircraft behavior.

C. SUMMARY OF CHAPTERS

Chapter II provides an overview of NPSNET. Chapter III discusses expert systems. Chapter IV discusses P-3 aircraft modeling techniques and aircraft dynamics. Chapter V explores aircraft ordnance expenditures. Chapter VI examines the expert system. Chapter VII discusses DIS networking and interface with NPSNET. Chapter VIII provides a summary of conclusions and further work. Appendices A and B list C source code for two expert system functions. Appendix C list the CLIPS source code for the expert system shell.

II. OVERVIEW OF NPSNET

NPSNET is a low-cost, real-time, three-dimensional visual simulation system, currently under development by researchers in the Computer Science Department at the Naval Postgraduate School (NPS) in Monterey, California [Zyda92]. NPSNET is designed to run on Silicon Graphics, Inc. IRIS workstations, a powerful family of commercial off-the-shelf (COTS) graphics workstations.

NPSNET is a totally interactive battle simulation system in which the user can select any one of 500 different active vehicles and control it with several devices, including a six degree of freedom SpaceBall, a keyboard, a mouse and a button/dialbox. Other vehicles in the simulation are being controlled by users on other workstations, expert systems, or by NPSNET itself.

For communication and interaction between local workstations, NPSNET broadcasts locally designed packets on an Ethernet network. For large scale interaction at many different levels, a translator has been implemented which provides the capability to transmit packets compatible with the Simulation Networking (SIMNET) protocol [Pope89]. Current work includes the design of an expanded translator which is compatible with the Distributed Interactive Simulation (DIS) protocol [IST91].

Vehicles and objects in NPSNET are modeled using the locally developed NPS Object File Format (NPSOFF) [Zyda93]. NPSOFF is an ASCII formatted language which incorporates many graphics library (GL) [SGI91] function calls into a single object file. The overall format of NPSOFF closely resembles a series of standard GL calls. By representing objects in this way, NPSOFF provides a simple method of encapsulating objects which are easily transportable between programs and can be referenced in an abstract manner. An ASCII format also makes the file readable and modifiable with any text editor. Ongoing work on NPSNET models includes implementation of the Graphics Data Language (GDL) which is a C++ based system to further encapsulate models and their properties [Wils92].

Most of the prior research for NPSNET has involved land-based units with only limited work focused on ocean going vessels and maritime aircraft. This limitation restricts the NPSNET battle simulation to primarily land-oriented operations. Inclusion of maritime aircraft and surface and subsurface vessels enables NPSNET to begin expanding its battle simulation program to include naval units.

III. EXPERT SYSTEMS

A. DESCRIPTION

An expert system is described as "a program that contains a large body of knowledge concerning one special field, this having been provided by one or more human experts in that field, and able to achieve the same performance in problem-solving as those experts" [Watt92]. Knowledge provided by these experts can be taken from written papers and resource material or, as is frequently the case, directly from the experts themselves. Much of the knowledge must be obtained through extensive personal interviews and discussions with the expert, since most expertise is embedded in layers of subliminal knowledge. The task of the expert system designer is to effectively model the expert's knowledge and embed it into the expert system.

B. DEFINITIONS

Data or information contained in an expert system can be divided into two major categories. Basic data that is not subject to interpretation, such as *the car is red*, is commonly known as **facts**. Logical data which requires some interpretation and a resulting action, such as *If the car is red, then I will buy it*, is known as **rules** or the **knowledge base**. Rules can be separated into two basic parts. The part between the *if* and the *then* is known as the left hand side of the rule. It contains a list of conditional facts. The part following the *then* is called the right hand side of the rule. It contains a list of actions that will be completed if all of the facts are true.

The **inference engine** is the part of the expert system which combines the facts and rules together and searches for any pattern matches between the two sets. A vast spectrum of inference engines has been developed for such fields as medicine, air traffic control, military applications, chemistry, space, and image processing. Some of the more famous are EXADS, AESOP, CERT, AIRPLAN, CRIN, PEER and HORSES [Watt92].

The inference engine is typically written in a higher level computer language. The most common languages in the Artificial Intelligence world are LISP and PROLOG, but expert systems are increasingly being designed using such languages as C, PASCAL, BASIC, PL/1, ADA and even FORTRAN. [Watt92]

C. CHAINING

Expert systems can follow two different strategies known as forward chaining and backward chaining. Forward chaining is the process of reasoning from known facts to the resulting conclusions. Backward chaining is the process of reasoning from known conclusions to the facts that caused them.

A simplified example of forward chaining is going to work in the morning. The desired result is to arrive at work. The commuter knows that if he has his keys, he can drive his car. If he can drive his car, he can get on the highway. If he follows the highway, it will take him to the building he works in. If he parks his car and goes into the building he works in, he has arrived at work and reached his goal. He knew the desired result and looked for a chain of facts to get him there.

An example of backward chaining is troubleshooting a car that will not start. The mechanic knows the symptoms, but does not necessarily know the cause. He may first check the condition of the battery. If it is good, then he might check the condition of the spark plugs. He continues checking parts, or facts, until he finds one that might cause the car to not start. He knows the result, the car will not start, and must backtrack, looking at all available facts to see which one caused the result.

When developing an expert system, one must determine which type of chaining is best suited for the problem to be solved. In some cases, both forward and backward chaining may be applicable, but typically one type is dominant. The developer must then choose an inference engine which optimizes the dominant chaining method.

D. CLIPS

The C Language Integrated Production System (CLIPS) is an inference engine designed by NASA which is rapidly gaining popularity in the expert systems world. With versions written both in C and in ADA, CLIPS can run on a variety of different platforms including UNIX, MS-DOS and MacIntosh. Knowledge in CLIPS is represented as facts, rules, functions and objects. [Giar91]

CLIPS is designed to be readily interfaced with other higher level languages such as C and ADA. There are two basic approaches when interfacing CLIPS with other languages. The user can write a higher level expert shell in CLIPS and call lower level functions written in the other language. Alternatively, the user can write the main driving program in the higher level language and call the CLIPS shell for any decision making. There seems to be no major advantage of one approach over the other.

E. CLIPS VERSUS PROLOG

Both CLIPS and PROLOG are readily available at NPS and have been successfully integrated with C. While CLIPS and PROLOG can each perform both forward and backward chaining, PROLOG is designed primarily to support backward chaining [Rowe88] and CLIPS is designed primarily to support forward chaining [Giar91]. A detailed rationale for the selection of CLIPS as the expert shell for this research is presented in Chapter VI.

IV. AIRCRAFT MODELING AND MOTION DYNAMICS

Previous research in aircraft flight control models in NPSNET ranges from fundamental helicopter motion to a detailed flight simulator for high performance aircraft [Cook92]. The primary scope of this work was to prove the concept of an expert system which broadcasts information over a network to NPSNET. Instead of developing a precise flight simulator for the P-3 aircraft, a simple aircraft motion model was developed which incorporates basic P-3 flight characteristics without loss of generality.

A. GRAPHICAL MODEL DEVELOPMENT

1. NPSOFF Format

While developing the graphical model for the P-3, the primary goal was to develop a model which could easily be transported into other software packages with little or no software modification. NPSOFF provided the framework to accomplish this goal. Due to the complexity of the shape of the P-3 aircraft, initial versions of the model were written using standard IRIS graphics library calls, allowing for easier modifications while fine tuning aircraft shape and colors. When the final model using standard library calls was completed, a simple conversion program was written in C which translated the standard drawing commands into NPSOFF commands and wrote them to a file. This final ASCII file was then ready to be used as an NPSOFF object, which allows the P-3 model to be easily transported to other pieces of software with little modification. Figure 1 illustrates the completed model.

2. Propeller Simulation

With a true spinning propeller, the viewer is able to see through the plane formed by the rapidly spinning propeller. The viewer can also sense that some material is present in the plane. Other visual clues within the propeller plane alert the viewer that motion is taking place. To simulate aircraft propeller motion, two approaches were explored. The

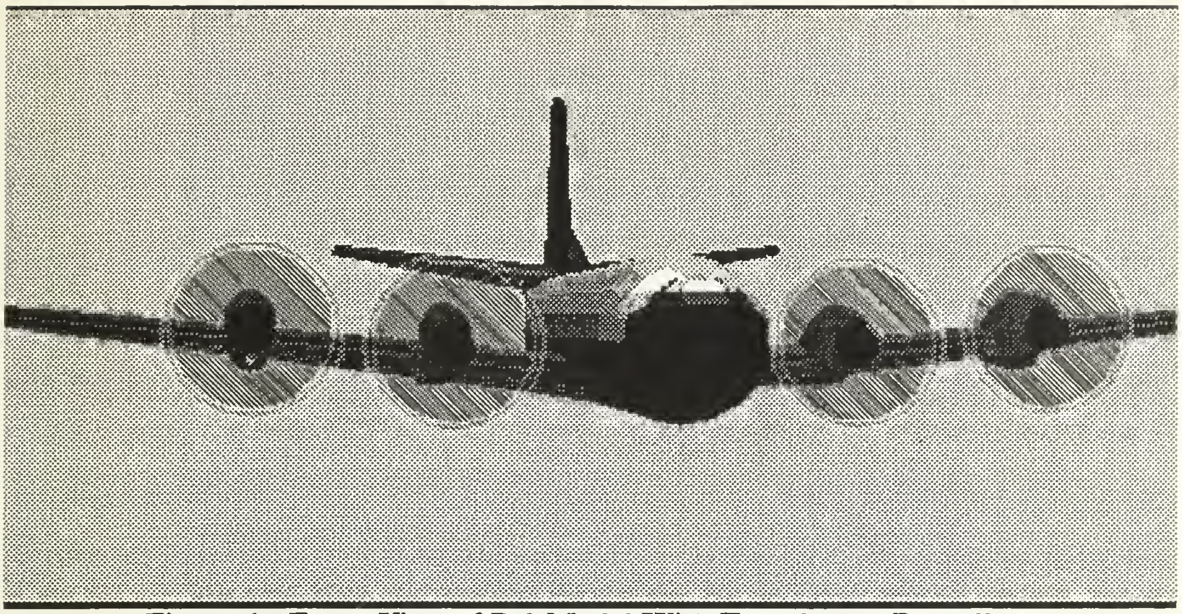
Figure 1 - Front View of P-3 Model With Translucent Propellers

first consisted of actually rotating models of propellers on the aircraft while the second utilized translucent discs in place of the spinning propellers.

a. Spinning Propeller Models

In the first approach, an NPSOFF model of a single four-bladed propeller was constructed, Figure 2. During the display loop, four copies of the propeller were drawn, each in the correct position in front of the corresponding engine nacelle. In each successive loop, each propeller was rotated slightly from the previous loop. The resultant image displayed four rotating propellers on the aircraft. Although this technique produced a sense of propeller motion, the speed of rotation was much slower than true propeller rotation, giving the impression of artificiality and actually detracting from the appearance of the model. Running the model on different levels of machines caused the propellers to rotate at different speeds, adding to the feel of artificiality. Additionally, adding the motion of the propellers required extra driving software to be incorporated into any software using the model. Figure 3 illustrates the static propellers attached to the P-3 model.

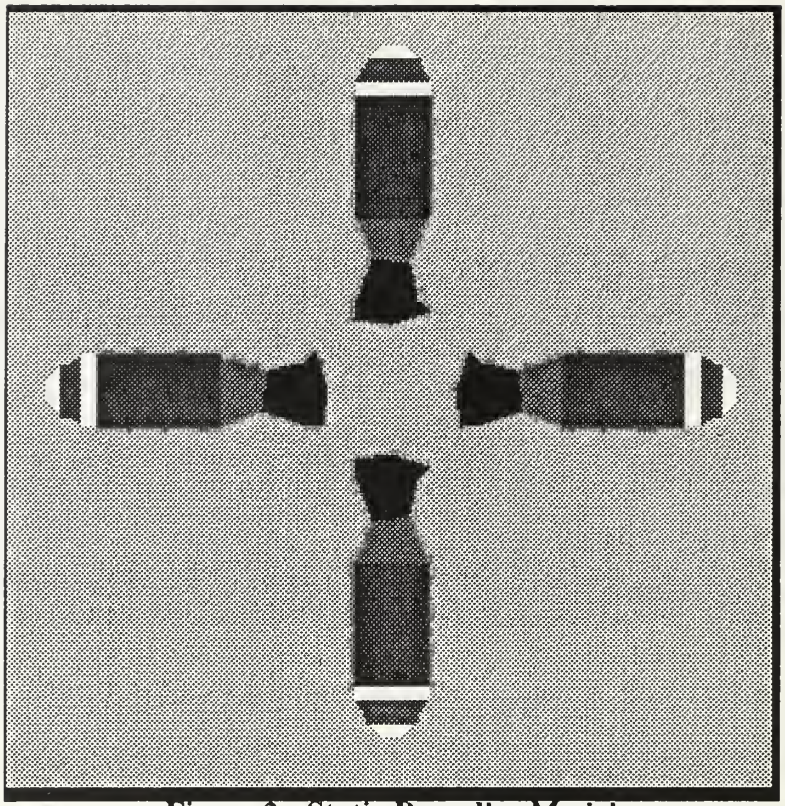
Figure 2 - Static Propeller Model

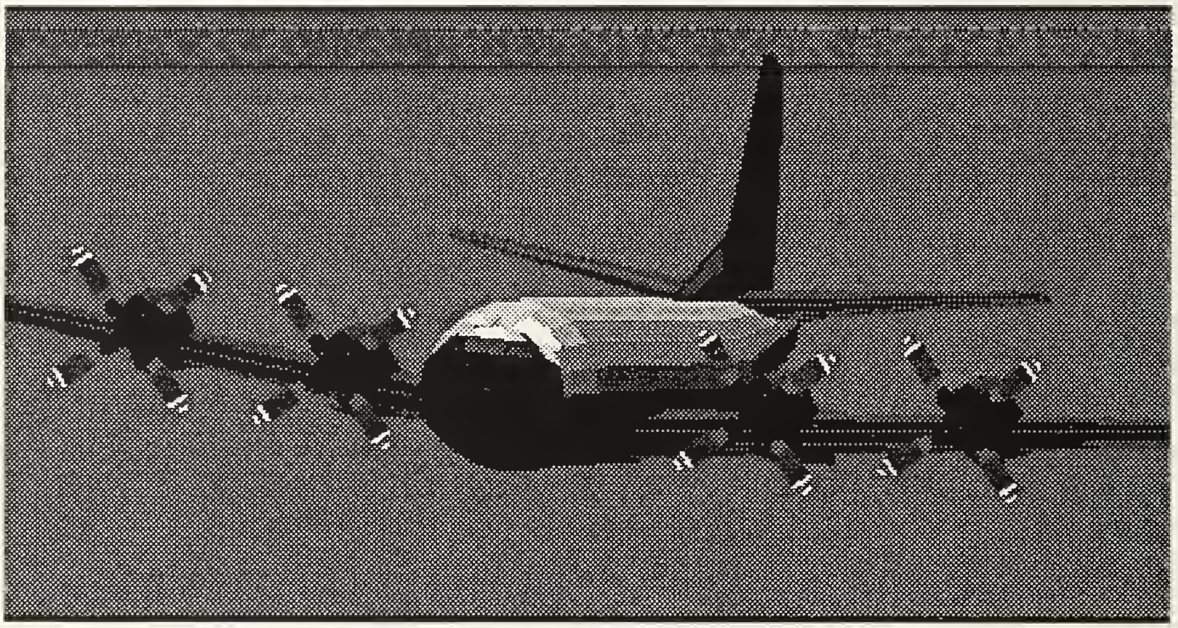
Figure 3 - Front View of P-3 Model With Static Propellers

b. Translucent discs

In the second, more effective approach to modeling spinning propellers, a grey translucent disc was created with a radius equal to the length of a single propeller blade. The outer edge of the disc was made up of alternating red and white translucent rings to simulate the safety markings on a spinning propeller. Setting the alpha value to 0.3 for the propeller material provided the desired translucency. There were several advantages to this method over the first one. Most importantly, the discs more closely resembled rapidly spinning propellers. Although the discs are fixed objects, when the aircraft is moved within the graphics environment, slight aliasing along the polygon edges within the discs creates the illusion of propeller motion. This unexpected side effect actually enhances the model. Since the discs are stationary objects relative to the aircraft, they can be made a permanent part of the total aircraft model without requiring embedded propeller driving software. Figure 1 illustrates the translucent discs mounted on the aircraft model.

3. Aircraft Lighting

An effective method of modeling exterior aircraft anti-collision and position lighting was developed through a series of experimental trials. The first approach consisted of setting the emission value of two small crossed polygons to the highest value available. At certain viewing angles, this produced the desired effect but, as the viewing angle changed, so did apparent light intensity. As the viewing angle relative to the plane of the light emitting polygon gets smaller, the cross section of a polygon becomes smaller. The result is a lower apparent light intensity. At very low angles, the apparent light was totally diminished. A second attempt used small light emitting cubes which produced results similar to the crossed polygons. The most effective method was obtained by drawing a small cluster of light emitting point sources. For each aircraft light, eight points were arranged to form the vertices of a cube and a ninth point was placed in the center of the cube. The distance between each point was close enough to give the appearance of a single, larger light source, except when viewed at very close distances. This technique produced a realistic light intensity, regardless of the viewing angle. Colors studied include red, green, and white, standard colors for aircraft position and anticollision lighting. Blinking lights were simulated by alternately drawing and then not drawing the light cluster.

B. AIRCRAFT MOTION CONTROL

1. Background

To control a typical aircraft, the pilot has a limited number of flight controls. To control aircraft speed, he has a power control and to control aircraft attitude he has a control stick or yoke. He must apply power to increase airspeed and move the control stick to change the attitude of the aircraft. To control the direction the aircraft moves through the air, the pilot only has to set an aircraft attitude. Once an attitude is set, the laws of aerodynamics change the aircraft's heading and position. For example, to turn the aircraft around its vertical axis, the pilot must rotate the aircraft around its longitudinal axis. This change in aircraft attitude modifies the lift vectors which in turn act upon the aircraft to move it around the vertical axis. Rudder control is used primarily to maintain coordinated or balanced flight.

To model aircraft motion for this study a set of motion control parameters was developed which cause the aircraft model to move in response to external stimulus, similar to the way a pilot controls the real P-3. To get the model from point A to point B, the speed must be set and the attitude of the model must be modified as necessary to allow the motion control equations to move it to the desired location. A variety of devices which control aircraft attitude and power were examined and are discussed below.

2. Assumptions

As stated earlier, the focus of this work was on expert systems vice precise flight simulation. The motion control equations were designed to realistically maneuver the model within the normal flight envelope of the P-3 aircraft, without requiring time consuming calculations of tedious aerodynamic equations. The motion control system was based on the following assumptions:

a. The model will remain within the normal P-3 on-station operating envelope which is:
Airspeed: 150 - 300 knots
Angle of Bank: < 70 degrees
Pitch: < 25 degrees nose up or down [NTPS83]

b. Once set, airspeed will remain constant, regardless of angle of bank or aircraft attitude. In other words, airspeed will not bleed off in climbs or turns or increase in descents.

c. Flight is assumed to be balanced at all times. Rudder control was ignored.

d. Landing gear and wing flap operations were not modeled. These devices are not normally used during ASW operations.

3. General Flight Maneuvers

To maintain consistency in the speed of the motion from one graphics platform to another, all motion computations are based on the system real time clock. Each time through the graphics loop, current system time is read and changes in aircraft attitude and position are calculated as a function of the time difference from the previous time through the loop. This method is known as Euler's method which computes a system's derivatives at some time t_k and updates the data structures for some time t_{k+1} based on those derivatives and the time difference, $t_{k+1} - t_k$ [Barz92].

Synchronizing time to motion is essential for network operations. If two machines on the network are operating at different CPU speeds, they must be able to move the model at the same speed during the dead reckoning time period between receipt of network updates. If the dead reckoning is based on frames instead of time and the machine speeds are different, the resulting image will appear to jump from position to position as new updates received from the sending station will not closely match the dead reckoning computations done by the receiving station.

4. Change in Coordinate Position

Computations for change in coordinate position between cycles in the graphics loop are relatively simple. Instantaneous aircraft velocity is maintained in a three element array with each element containing the x, y and z component of the velocity. Once the time difference between loops is determined, the next position coordinates are computed as follows:

$$x_{k+1} = x_k + v_x \times \Delta t \qquad \text{(Eq 1)}$$

$$y_{k+1} = y_k + v_y \times \Delta t \qquad \text{(Eq 2)}$$

$$z_{k+1} = z_k + v_z \times \Delta t \qquad \text{(Eq 3)}$$

Where:

x_{k+1}, y_{k+1}, and z_{k+1} are the next position coordinates.

x_k, y_k, and z_k are the previous position coordinates.

v_x, v_y and v_z are the x, y, and z components of current aircraft velocity.

Δt is the time difference between loops.

Note that the position is based on instantaneous velocity only. Change in position is not dependent directly upon the orientation the aircraft. Only its current acceleration and velocity are factors.

5. Turning flight

Changes in aircraft orientation during turning flight are a function of input flight control stimuli and aircraft speed. Flight control stimuli, discussed below, control the model's motion around the x-axis (roll) and the z-axis (pitch). Aircraft roll and speed determine the model's change in orientation around the vertical y-axis (yaw or heading) as follows:

$$\Delta y_{rot} = -\Delta t \sin(roll) \times v \times K \qquad \text{(Eq 4)}$$

where:

Δy is change in aircraft heading.

Δt is the time difference between frames.

$roll$ is the angle of bank of the aircraft.

v is aircraft velocity in knots.

K is a modification factor for the P-3 based on speed.

Note: K is an approximation which must be mutliplied by the velocity to achieve a proper turn radius. The turn radius is based on the rule of thumb that a standard rate turn (3 deg/sec) is equal to approximately 10 percent of the aircraft's airspeed times two. For example, at 150 knots, 30 degrees angle of bank produces a 3 degree per second turn. Figure 4 shows the graph used to convert the rate of turn into realistic values. The precision of this chart is based purely on pilot experience factors.

6. Rendering the Model Using Euler Angles

A common technique for parameterization of orientation space is known as Euler angles where total rotation is described as a sequence of rotations around the three axes [Watt92]. One problem encountered with Euler angles is known as gimbal lock where rotation of $\pi/2$ radians on one axis will make rotation around a second axis ineffective. Since the modeled flight envelope of the P-3 is limited to well below $\pi/2$ radians in the roll and pitch axes, implementation of Euler angles was an acceptable approach.

The order of rotation is critical when using Euler angles [Watt92]. A different order of rotation will produce a different final position. For this research, the order of rotation was chosen to be roll, pitch and yaw, a convention commonly used by aeronautical engineers when representing aircraft orientation.

7. Flight Controls

Several input devices to simulate aircraft flight controls were implemented and tested. The advantages and disadvantages of each device tested are discussed below.

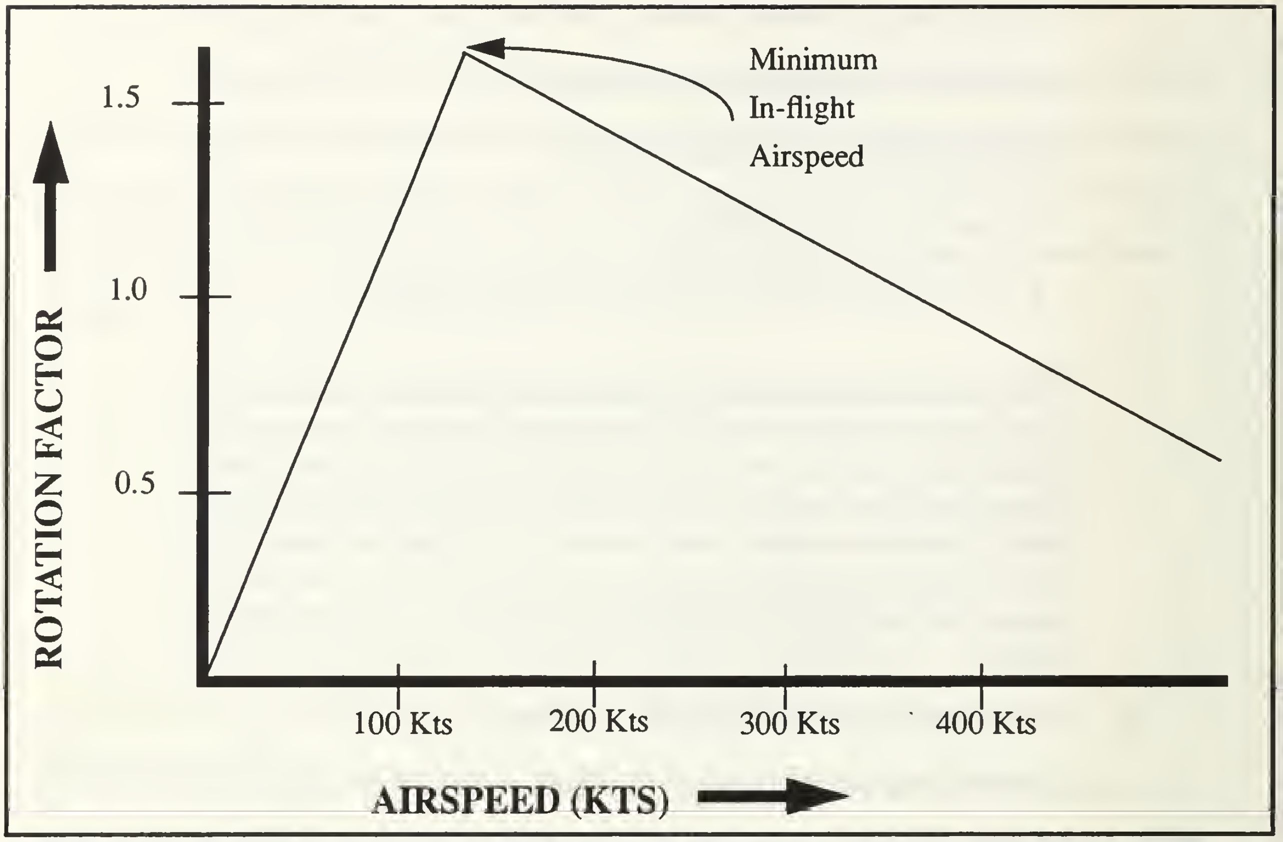

Figure 4 - Change in Heading Factor

a. Joystick

The joystick control provided the most natural user interface with the system. For this research, a two-piece device providing both a joystick and a power control was utilized, Figure 5. Speed of the model was controlled by the power control and roll and pitch were controlled by the joystick. Increasing the magnitude of joystick motion increased the rate of change on the respective axis. When the user changes model attitude, the motion equations change model motion.

b. Mouse

The mouse provided an acceptable alternative to the joystick, since a mouse is readily available on all graphics systems. Using a display panel as shown in Figure 6, the operator uses the mouse to move the spots on the display. The position of the spots is

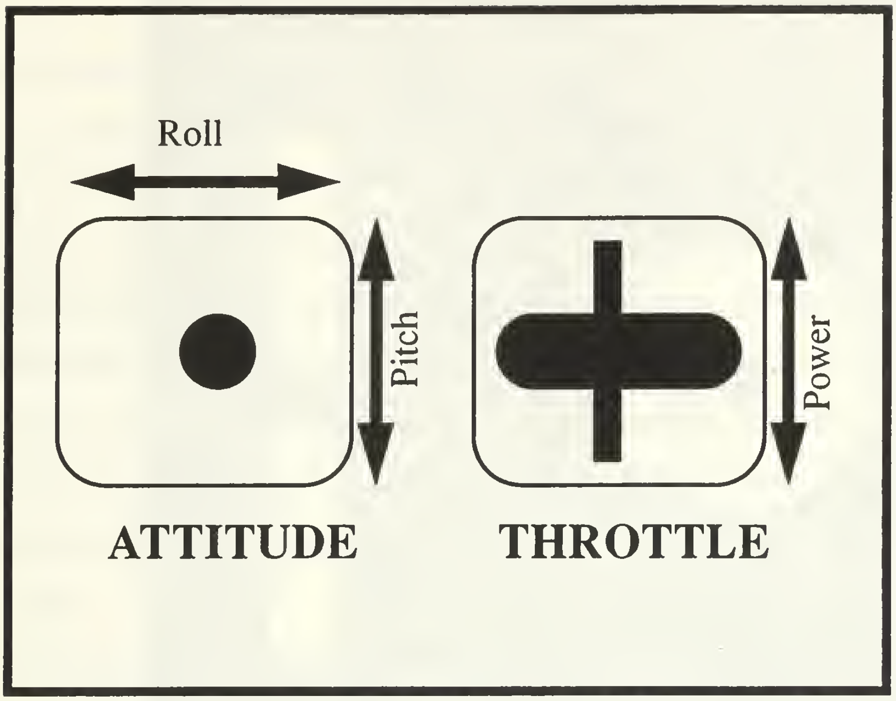

Figure 5 - Block Diagram of Joystick Controls

directly related to the position of a joystick when looking down from the top. A simple conversion translates spot coordinates into joystick position to control the model.

c. *Keyboard*

Utilization of the keyboard to control the model was investigated but not implemented. Using keys to control rate of rotation was not intuitive and proved confusing for most users. Without extensive training and practice, it is difficult for most users to remember which keys perform which functions. All graphics workstations are provide with a mouse control which proved to be a much better alternative to the keyboard.

d. *Ascension Bird*

The Ascension Company markets a three dimensional mouse known as the Ascension Bird. It provides three dimensional mouse position and also orientation around

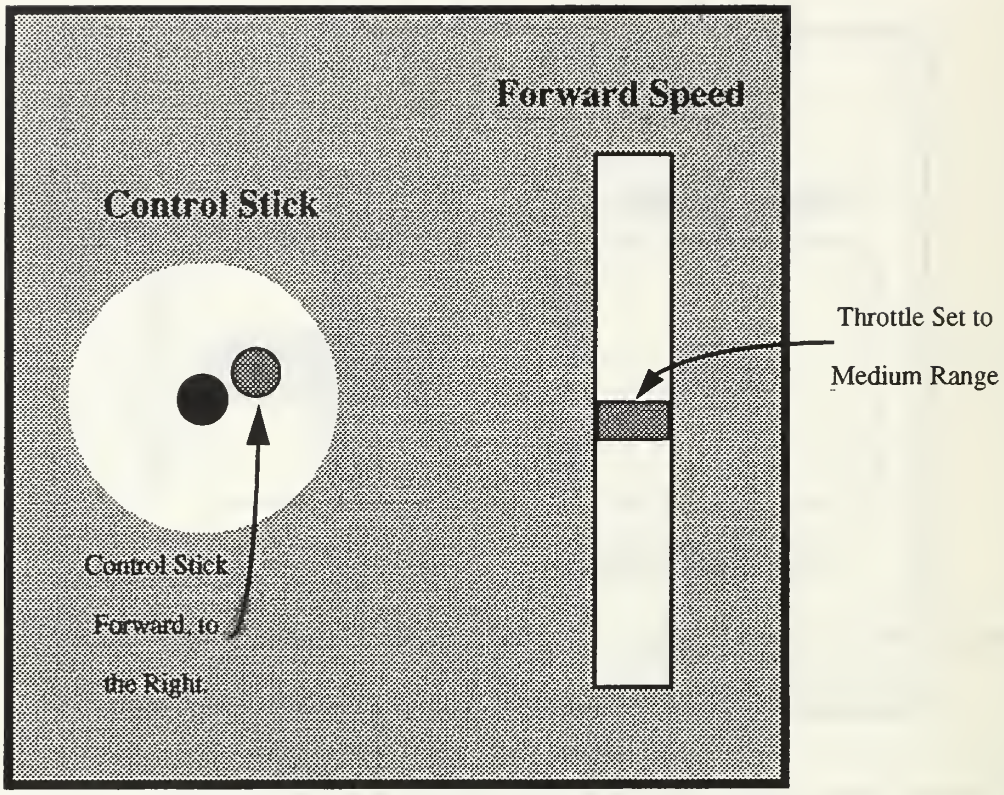

Figure 6 - Mouse Control Panel

the three orthogonal axes. To implement the Ascension Bird, the motion model was modified slightly to follow a *scene in hand metaphor* [Ware90]. The scene in hand metaphor moves the object being displayed in the exact manner that the Bird is being moved. For example, if the Bird is turned clockwise, the object turns clockwise. To model the scene in hand, the computations for changes aircraft roll, pitch, and yaw were replaced with the current physical orientation of the Bird. This allowed the user to instantly set an aircraft attitude by simply changing the orientation of the Bird. Actual aircraft forward motion was still computed based on instantaneous aircraft attitude. Changes in heading followed changes in Bird heading.

The Ascension Bird provided a unique approach to controlling the model, but proved to be less effective than the joystick for several reasons. Spurious noise caused the model to be quite unstable, especially as the Bird approached the fringes of its signal range. Also, the Bird allowed the user to readily maneuver the model outside of the normal operating envelope of the P-3, providing a feeling of artificiality.

An annoying side effect of the scene in hand metaphor occurs when using the Bird to control heading: the user gets "wrapped up" by the control cable when turning more than 360 degrees. We tried a small software change to convert the model to a *flying vehicle control* [Ware90] metaphor. By allowing the Bird to control only roll and pitch, this metaphor more closely resembled joystick control. However, the flying vehicle metaphor using the bird was still judged to be undesirable Although the flying vehicle metaphor gives the user a more realistic feeling when controlling of the model, he soon experiences fatigue from holding the Bird at the desired attitudes. He cannot let go of the controls like he can with the joystick.

e. SpaceBall

The SpaceBall, produced by Interactive Graphic Techniques, is an isometric joystick which receives input through pressure sensors on the SpaceBall vice position of the ball. Additionally, an extra degree of control is provided by twisting the ball in a forward or backward direction. Motion equations are similar to those used by the joystick. Many users were not satisfied with the feel of the SpaceBall. Lack of motion of the SpaceBall itself seemed unnatural compared to the joystick. Also, users inexperienced with the SpaceBall easily confuse forward ball pressure with ball rotation and find it difficult to overcome.

V. AIRCRAFT ORDNANCE MODELING

A. BACKGROUND

The P-3 aircraft is capable of carrying a large variety of weapons and sensing devices which are dropped into the water. These include, but are not limited to sonobuoys, torpedoes, mines, depth bombs, flares, smoke markers, rockets and missiles. The type of mission determines which devices are carried on any given flight. During an ASW prosecution, sonobuoys are used to search for the submarine and torpedoes are used for destroying the submarine. Since this study deals primarily with the ASW mission, only sonobuoys and torpedoes were modeled. [NTPS83]

B. BASICS OF PROJECTILE MOTION

1. Position and Velocity Computations

The dynamics of objects released from an aircraft follow the basic principles of projectile motion. At the time of release from an aircraft, an object assumes an initial velocity. The two major forces that act upon a falling projectile are gravity and air friction. One of the simpler methods of iteratively computing projectile motion is Euler's method [Finn90]. Euler's method breaks a large interval into a set of smaller sub-intervals. Sequentially, at the starting point of each sub-interval, it uses the derivative of a function to determine the approximate value of that function at the end point of each sub-interval. Since acceleration is the first derivative of velocity and the second derivative of position, the Euler method can be applied directly to motion computations using the following equations:

$$v_{n+1} = a\Delta t + v_n \qquad \text{(Eq 5)}$$

$$x_{n+1} = \frac{1}{2}a\Delta t^2 + v_n t + x_n \qquad \text{(Eq 6)}$$

When applying the Euler method to computer graphics, Δt in the above equations is replaced with the time interval between drawing cycles or frames. Actual computations are performed on the x, y, and z components individually.

2. Acceleration

As mentioned above, the two major forces acting on a falling body are gravity and air friction. The acceleration due to gravity, g, is a constant 32.174 ft/sec^2 [Weas66] acting downward along the y axis. Drag is the force created by the friction of air against the body as it falls. The direction of drag is opposite the direction of motion. The magnitude of drag is dependent upon the velocity of the body and is computed as follows [Hall88]:

$$|D| = \frac{1}{2}C\rho A v^2 \qquad \text{(Eq 7)}$$

Where:

C is the Coefficient of Drag.

ρ is the density of air.

A is the cross sectional area of the object.

v is the velocity of the object.

Drag and gravity are then incorporated into the following equations to compute the total acceleration for each axis:

$$a_x = -sign(v_x)Kv_x^2 \qquad \text{(Eq 8)}$$

$$a_y = -g - sign(v_y)Kv_y^2 \qquad \text{(Eq 9)}$$

$$a_z = -sign(v_z)Kv_z^2 \qquad \text{(Eq 10)}$$

Where:

a_x, a_y, a_z are the accelerations in the x, y, and z axes.

$K = \frac{1}{2}C\rho A$

v_x, v_y, v_z are the velocity components along the x, y, and z axes.

Sign(x) is a function which returns 1 if x is positive or -1 if x is negative. Since the original sign of the velocity is lost when it is squared, the *sign* function is required to ensure that the correct direction of the drag component on each axis is maintained.

3. Orientation Computations

The aerodynamic shape of the ordnance used in this study forces the falling bodies to point in the direction of motion. Only the y (yaw) and z (pitch) axes are effected by this phenomenon. To reproduce this orientation correctly, it is necessary to determine the amount of pitch and yaw which must be applied to the object when rendered. Applying simple trigonometry to the orthogonal components of the velocity vector, these angles can be determined with the following set of equations:

$$v_{xz} = \sqrt{(v_x)^2 + (v_z)^2} \qquad \text{(Eq 11)}$$

$$Rot_z = \operatorname{atan}\left(\frac{v_y}{v_{xz}}\right)\frac{180}{\pi} \qquad \text{(Eq 12)}$$

$$Rot_y = \operatorname{atan}\left(\frac{v_z}{v_x}\right)\frac{180}{\pi} \qquad \text{(Eq 13)}$$

Where:

v_x, v_y and v_z are the x, y, and z components of the velocity.

v_{xz} is the vector sum of v_x and v_z.

Rot_y is the rotation around the y axis in degrees.

Rot_z is the rotation around the z axis in degrees.

C. TORPEDOES

1. Description

The P-3 is capable of carrying up to eight MK-46 torpedoes in the bomb bay, just forward of the wings. Each torpedo is hung onto a rack and stabilized with four adjustable braces. The braces screw down tightly against the torpedo, preventing it from rocking. When the torpedo is launched, an electrically operated solenoid releases the two hooks and the torpedo simply falls out of the rack. The pressure of the braces acts like a spring to give the torpedo a small initial downward velocity when the hooks are released. However, this force is negligible and can be ignored. Gravity is the overriding force during torpedo launch.

The torpedo is oriented in the bomb bay with the nose pointed forward. When the torpedo is launched, it is already pointed into the wind and its long cigar shape keeps it oriented into the direction of travel. Immediately after clearing the aircraft, a small stabilizing parachute is deployed from behind the torpedo's fins. The parachute acts to slow the torpedo, causing its nose to point slightly downward. This forces a higher water entry angle and prevents the torpedo from skipping off of the water's surface. Figure 7 shows the aircraft model releasing a torpedo. For this study, only the in-flight characteristics were modeled for the torpedo. The effects of the parachute were modeled mathematically. The effect of the parachute on the visual model is negligible and therefore was ignored.

After water penetration, the torpedo begins a search maneuver which is classified and beyond the scope of this study. For this simulation, the torpedo simply sinks to a preset depth and detonates, effectively modeling a depth bomb.

2. In-flight Characteristics

Modeling the in-flight characteristics of the torpedo was straight forward. Initial torpedo orientation and velocity were set equal to aircraft orientation and velocity at the time of launch. Each time through the graphics loop, the torpedo's instantaneous acceleration was updated using Equations (Eq 8), (Eq 9), and (Eq 10).

After computing acceleration for each axis, velocity and position were updated using the standard Euler motion Equations (Eq 5) and (Eq 6). To keep the torpedo oriented into the direction of travel, its orientation was computed using Equations (Eq 11), (Eq 12), and (Eq 13) with the velocity vector of the torpedo as the input parameter.

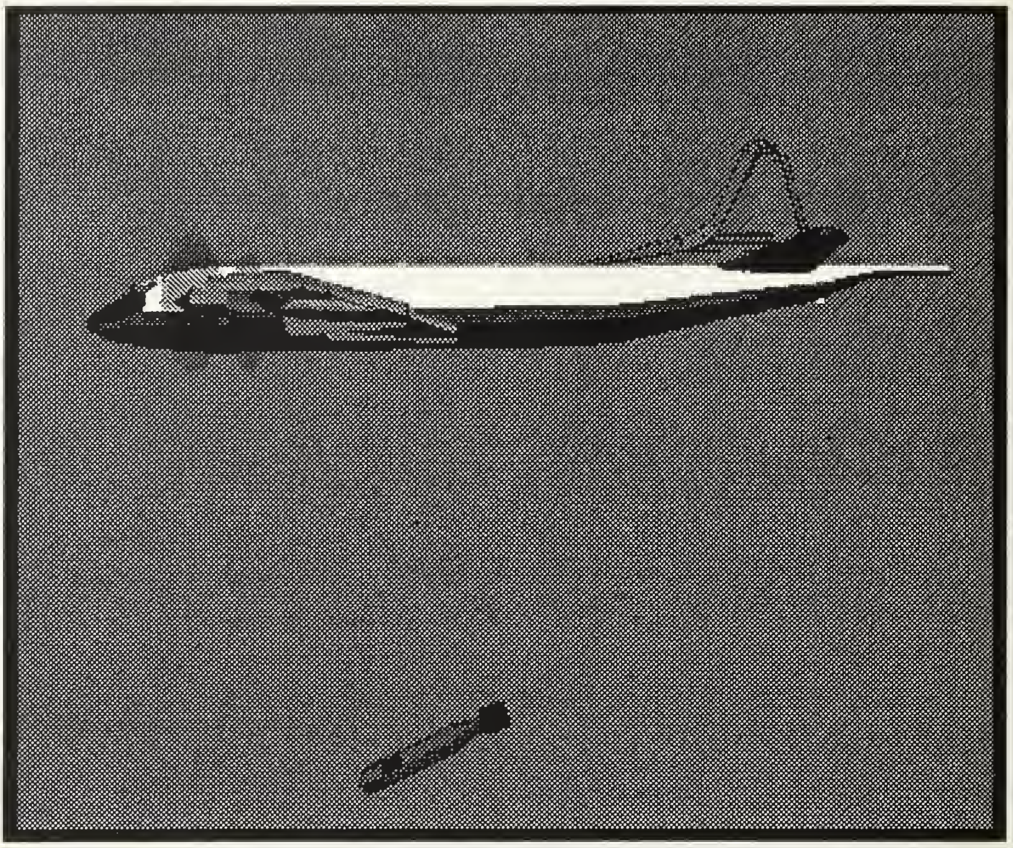
Figure 7 - Torpedo Drop By P-3 Model

Actual drag constant values of the torpedo were not readily available, however, after in-depth analysis of the behavior of the torpedo, we determined the overall drag constant to be approximately 0.04. Increasing the drag constant slightly provided a mathematical simulation of the stabilizing parachute's affect on the velocity of the torpedo.

D. SONOBUOYS

1. Description

The P-3 can carry up to 84 sonobuoys. The sonobuoy is a sensor used by the flight crew onboard the P-3 to listen for acoustic signals emitted by a submarine. Cylindrically shaped, it is approximately 40 inches long and 8 inches in diameter, Figure 8. Sonobuoys are launched out of a series of recessed tubes in the underside of the aircraft fuselage, just behind the trailing edge of the wings. To prevent collision with the aircraft, an explosive

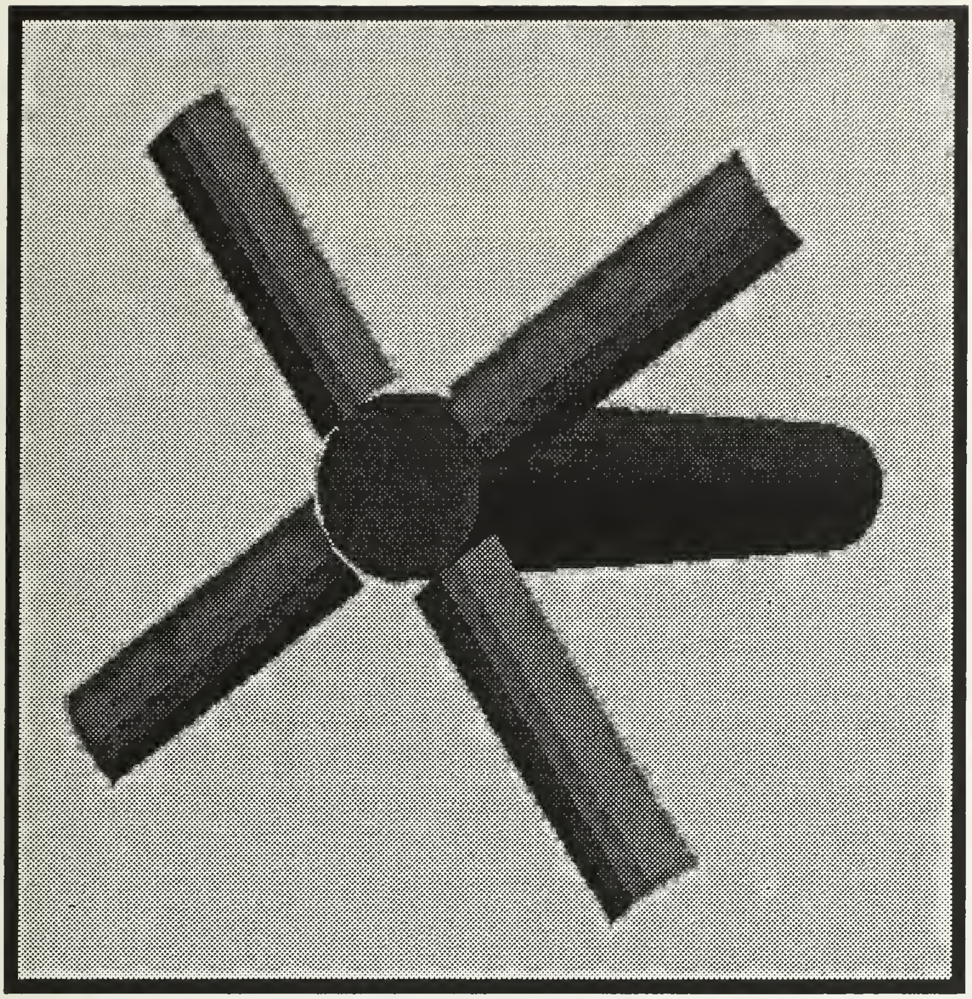
Figure 8 - Sonobuoy Model With Stabilizing Fins Open

cartridge is used to propel the sonobuoy into the airstream at an angle of about 60 degrees below the longitudinal axis aft of the aircraft, Figure 9. The exact initial ejection velocity of the buoy was unavailable, but experimentation determined that an initial value of 200 knots, relative to the aircraft, was realistic. Shortly after launch, spring loaded stabilizing fins open at the top of the sonobuoy, slowing it and rotating it into a more vertical position. Rotation about the longitudinal axis of the buoy is negligible. Upon water impact, the fins are ejected and a flexible antenna deploys. The sonobuoy floats on the water and deploys a hydrophone (underwater microphone) several hundred feet below it. Any acoustic signals

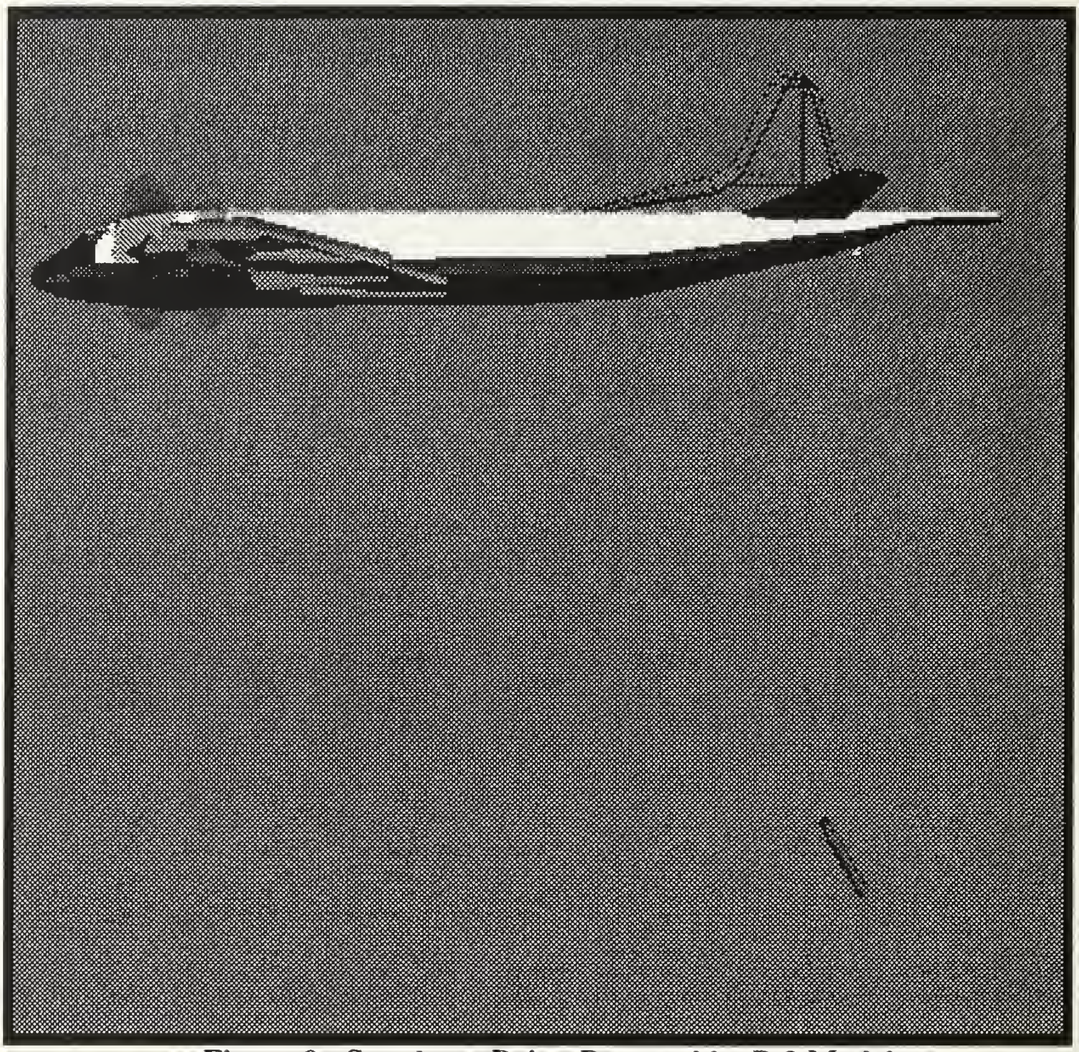
Figure 9 - Sonobuoy Being Dropped by P-3 Model

picked up by the hydrophone are transmitted back to the aircraft for analysis by the flight crew. Salt water activated batteries provide power for the electronics. Sonobuoys are designed to stay afloat from 30 minutes to eight hours. At the end of the period, it fills with water and sinks to the bottom. Some sonobuoy types can be scuttled by the flight crew.

2. In-flight Characteristics

Correctly modeling the in-flight behavior of the sonobuoy proved to be a much more challenging problem than the torpedo. In comparison with the torpedo which is dropped at aircraft speed and oriented in the direction of travel, the sonobuoy is launched with an initial velocity which is approximately 60 degrees down and aft of the aircraft. Its orientation is away from the direction of travel and, when the stabilizing fins begin to open, the model must be rotated into the direction of travel. To model variations in the delay time from launch until the beginning of fin opening, a standard random number generator was used. A side effect of this process provided a simulation for intermittent buoy failures by not opening the fins until after water impact. To solve the buoy modeling problem, it must be divided into two totally separate parts: motion computations and orientation computations. They are described below.

3. Motion Computations

The sonobuoy acceleration equations are similar to the torpedo acceleration equations except the coefficient of drag is a function of the opening angle of the stabilizing fins. The acceleration equations are:

$$a_x = -sign(v_x) K \sin\theta \qquad \text{(Eq 14)}$$

$$a_y = -g - sign(v_y) K \sin\theta \qquad \text{(Eq 15)}$$

$$a_z = -sign(v_z) K \sin\theta \qquad \text{(Eq 16)}$$

where:

θ = stabilization fin opening angle.

a_x, a_y, a_z = acceleration components along the x, y, and z axes.

g = acceleration due to gravity.

K = total coefficient of drag. (Actual drag coefficients were unavailable. For this study, K was set to 0.2.)

v_x, v_y, v_z= instantaneous buoy velocity along each axis.

sign() = function to determine sign of the velocity along a given axis.

In order to implement the standard Euler motion equations, it was necessary to determine the initial real world velocity of the buoy which was different from, but dependent upon aircraft velocity. With no aircraft rotation, the steady-state initial velocity of the buoy is the vector sum of the aircraft velocity along the x axis and the buoy ejection velocity, relative to the aircraft, Figure 10. To determine the real world velocity, the

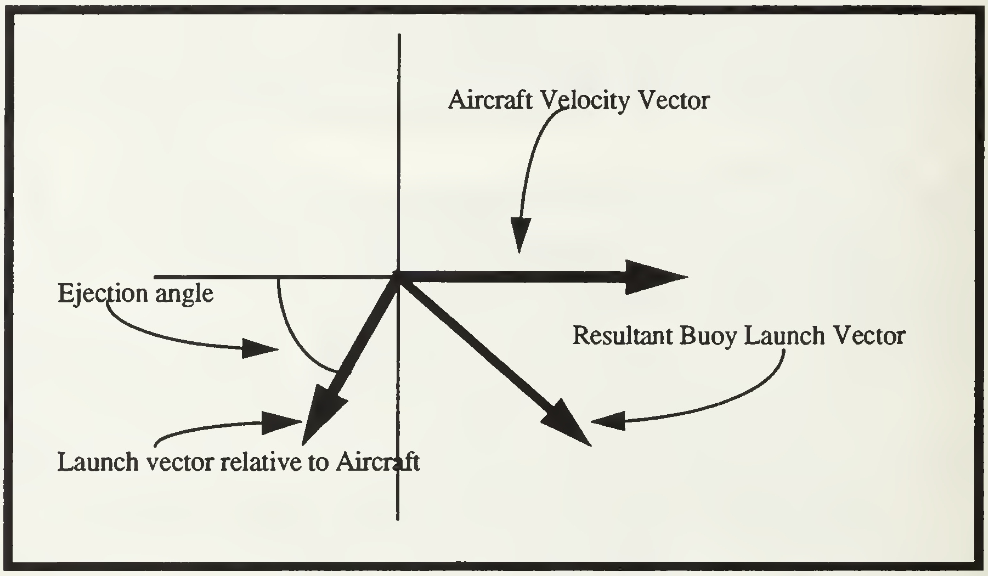
Aircraft Velocity Vector
Ejection angle
Resultant Buoy Launch Vector
Launch vector relative to Aircraft

Figure 10 - Addition of Relative Buoy Velocity to Aircraft Velocity

resultant steady-state buoy velocity vector must be mathematically rotated around the three orthogonal axes in the same order and magnitude that the aircraft has been rotated. The three matrix multiplications required to rotate the vector around the roll (x axis), pitch (z axis) and yaw (y axis) in sequence are as follows:

$$\begin{bmatrix} v_{wx} \\ v_{wy} \\ v_{wz} \end{bmatrix} = \begin{bmatrix} \cos\phi & 0 & \sin\phi \\ 0 & 1 & 0 \\ -\sin\phi & 0 & \cos\phi \end{bmatrix} \times \begin{bmatrix} \cos\alpha & -\sin\alpha & 0 \\ \sin\alpha & \cos\alpha & 0 \\ 0 & 0 & 1 \end{bmatrix} \times \begin{bmatrix} 1 & 0 & 0 \\ 0 & \cos\theta & -\sin\theta \\ 0 & \sin\theta & \cos\theta \end{bmatrix} \times \begin{bmatrix} v_x \\ v_y \\ v_z \end{bmatrix} \qquad \text{(Eq 17)}$$

Where:

v_{wx}, v_{wy}, v_{wz} = the world x, y, and z components of initial buoy velocity.

v_x, v_y, v_z = x, y, and z components of unrotated buoy initial velocity.

ϕ = aircraft rotation around the y axis.

α = aircraft rotation around the z axis.

θ = aircraft rotation around the x axis. [Fole87]

Note that the order of matrix multiplications is from right to left. The resultant of the rotations is the real world initial launch velocity of the buoy, broken into components along the three axes.

Using (Eq 14), (Eq 15) and (Eq 16) above and the real world initial buoy velocity found in (Eq 17), it is just a matter of plugging those values into the standard Euler motion equations to determine each successive buoy acceleration, velocity and position.

4. Orientation Computations

To compute the correct buoy orientation, the problem must be broken into three different phases. The first phase determines the orientation of the buoy immediately after launch. The final phase determines the orientation of the buoy after the fins are fully open. The middle phase covers the transition between the first and last phases.

a. Immediately After Launch

When the buoy is launched into the airstream its orientation is determined by the angle of the launch tube and the attitude of the aircraft. To determine the real world attitude of the sonobuoy at launch, a process similar to determining initial buoy velocity is used. The only difference is that aircraft velocity is not factored into the equation. The vector representing the angle of the launch tubes relative to the aircraft is shown in Figure 11. It is mathematically rotated around the x, z and y axes to match the aircraft rotations as follows:

$$\begin{bmatrix} Rot_x \\ Rot_y \\ Rot_z \end{bmatrix} = \begin{bmatrix} \cos\phi & 0 & \sin\phi \\ 0 & 1 & 0 \\ -\sin\phi & 0 & \cos\phi \end{bmatrix} \times \begin{bmatrix} \cos\alpha & -\sin\alpha & 0 \\ \sin\alpha & \cos\alpha & 0 \\ 0 & 0 & 1 \end{bmatrix} \times \begin{bmatrix} 1 & 0 & 0 \\ 0 & \cos\theta & -\sin\theta \\ 0 & \sin\theta & \cos\theta \end{bmatrix} \times \begin{bmatrix} -\cos(ej\theta) \\ -\sin(ej\theta) \\ 0 \end{bmatrix} \quad \text{(Eq 18)}$$

Where:

Rot_x, Rot_y, Rot_z = x, y and z components of the rotated vector

$ej\theta$ = Buoy ejection angle relative to the aircraft

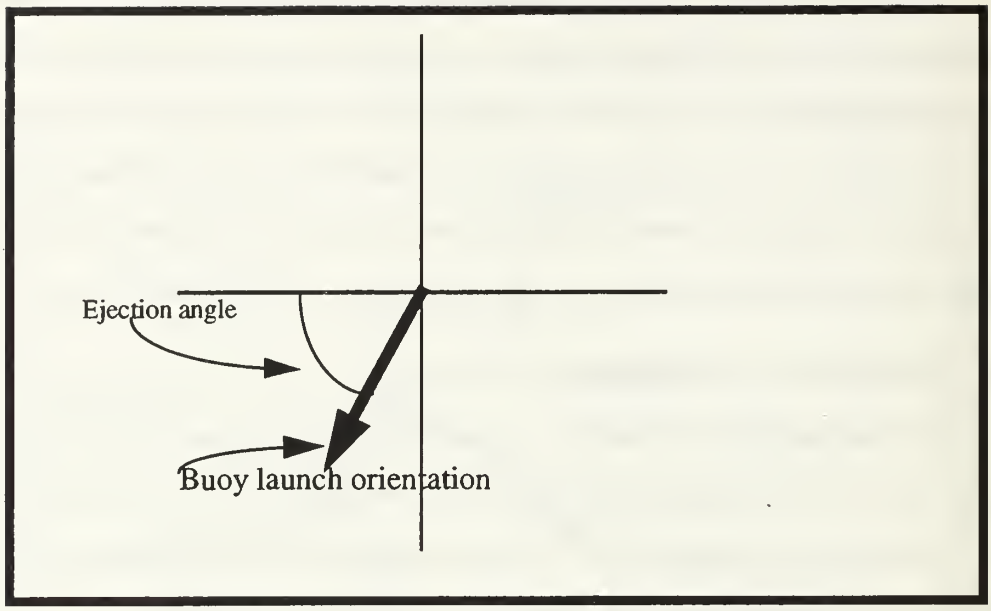

Figure 11 - Buoy Launch Orientation Vector

ϕ = aircraft rotation around the y axis

α = aircraft rotation around the z axis

θ = aircraft rotation around the x axis

As before, the matrix multiplications are from right to left. From the resultant vector, the angles of buoy orientation are determined using (Eq 11), (Eq 12), and (Eq 13). The buoy is rendered using those angles.

b. Transition Between Phases

When the fins start to open, the force of the drag on the fins causes the buoy to rotate into the direction of travel. To model this, the instantaneous buoy velocity is used in (Eq 11), (Eq 12), and (Eq 13) to compute an instantaneous target orientation for each axis based on the angle of velocity of the buoy when fin opening is initiated. The difference between the target orientation angles and the current buoy orientation angles is divided by 10 to get an incremental value. Each time through the graphics loop, the incremental value is subtracted from the original orientation until it reaches the target orientation angles. The result is a smooth rotation into the direction of travel. Figure 12 shows the algorithm for the transition. Correction factors are added to compensate for transitions through the ±180° heading boundary.

c. *Fins Fully Open*

When the fins are fully open, the force of drag on the fins pulls at the top of the buoy, opposite the direction of travel. This force, acting against the acceleration of gravity, causes the buoy to rotate downward to provide for a more vertical water entry. At this point, the buoy attitude is based solely upon the buoy velocity angle. Equations (Eq 11), (Eq 12), and (Eq 13) are used to compute the orientation angles for each axis until water impact.

```
The code below executes once when blades begin to open:
    /* Determine orientation we want to go to, based on current velocity */
    xz_v_zero = sqrt(buoy_v_zero[X] buoy_v_zero[X] +
buoy_v_zero[Z]*buoy_v_zero[z]);
```

```
The code below executes once when blades begin to open:
    /* Determine orientation we want to go to, based on current velocity */
    xz_v_zero = sqrt(buoy_v_zero[X] buoy_v_zero[X] +
buoy_v_zero[Z]*buoy_v_zero[z]);
```

```
    target_orient[Y] = atan2(buoy_v_zero[Z], buoy_v_zero[X]) * RAD_TO_DEG;
    target_orient[Z] = atan2(buoy_v_zero[Y], xz_v_zero) * RAD_TO_DEG;
    target_orient[X] = 0.0;

    /* Determine how far we have to go for each axis. */
     for (i=0; i<3; i++)
    {
      diff = target_orient[i] - buoy_orient[i];
      /* See if we cross */- 180.0 and compensate */
      if (diff < -180.0)
        diff += 360.0;
      else
        if (diff > 180.0)
          diff -= 360.0;
      /* Determine incremental value */
      increment[i] = diff/10.0;
      } /* end for (i=0; i<3; i++) */


The code below is embedded in graphics loop:
    done_flag = TRUE;
    /* Loop once for each axis */
    for (i=0; i<3; i++)
    {
      /* See if we have reached our goal yet */
      if (fabs(target_orient[i] - buoy_orient[i]) > fabs(increment[i]))
      {
        buoy_orient[i] += increment[i];
        /* Check for going through +/- 180.0 degrees */
        if (buoy_orient[i] < -180.0)
          buoy_orient[i] += 360.0);
        else
          if (buoy_orient[i] > 180.0)
            buoy_orient[i] -= 360.0;
      /* Set flag to indicate an axis was not done yet */
      done_flag = FALSE;
      } /* end if */
    } /* end for loop */
```

Figure 12 - Algorithm for Computing Buoy Rotation During Transition Phase

VI. THE EXPERT SYSTEM

A. OVERVIEW OF P-3 ASW SEARCH

When a P-3 crew searches for a submarine, it drops sonobuoys in a pre-determined pattern to cover the search area. The sonobuoys relay underwater acoustic information back to the aircraft where it is analyzed for submarine sound frequencies. Sonobuoys are either active or passive. Active buoys emit a sonar signal and wait for an echo from any large objects that may be in the area. If an echo is received, the crew can determine both bearing and range of the contact from the buoy. Passive buoys are listen-only devices. If the crew detects sounds from a submarine on a single passive buoy, it can determine bearing and sometimes approximate range to the target, depending on the amplitude of the sound received. Typically, when using passive buoys, the crew attempts to gain contact on more than one buoy and determines the submarine's location by calculating the intersection of bearing lines from different buoys to the submarine. By taking successive fixes on the submarine, the crew can determine its course and speed. Depending on mission requirements, the crew will either track the submarine or drop a torpedo on it to destroy it. For this system, to establish a proof of concept, we decided to implement only passive buoys. Expanding the system to include active buoy capability simply involves adding additional rules.

A typical P-3 mission carries twelve crew members consisting of three pilots, a tactical coordinator (TACCO), a navigator/communicator (NAV/COMM), three sensor operators, two flight engineers, an ordnanceman, and a technician. Either the senior pilot or the TACCO assumes the role of the mission commander who directs the ASW problem. The primary players in the tactical problem are the pilots, the TACCO and the sensor operators. Other crew members augment the mission by providing support services. The flow of tactical information within the flight crew is depicted in Figure 13. Data is shared between the tactical crew members and the mission commander, who makes decisions based on the data received and sends commands back to the crew. For example, he may ask the sensor

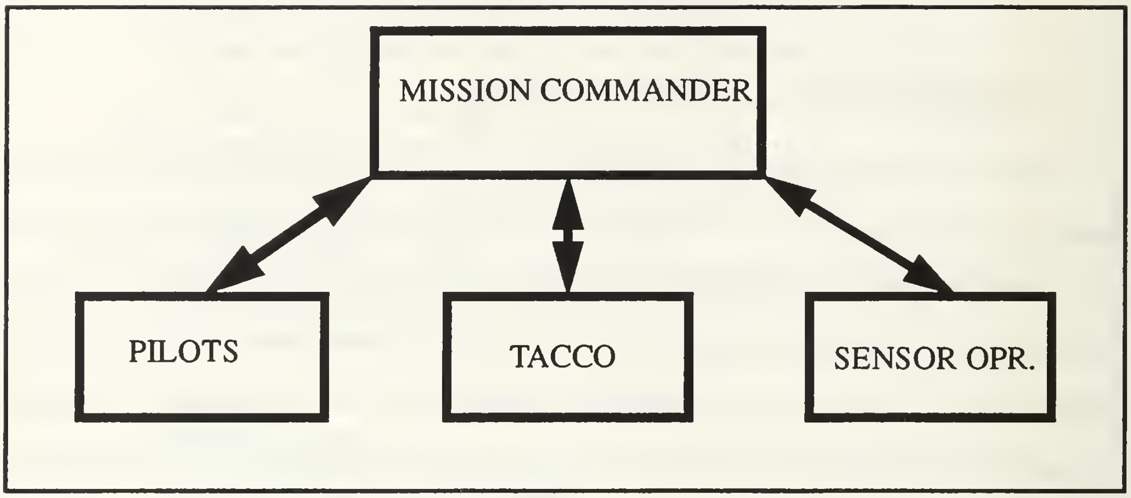

Figure 13 - Information Flow Between Tactical Crewmembers

operators for updates on contact information from the buoys; he may ask the TACCO to compute bearing intersections or to recommend new buoy positions; or he may direct the pilots to fly to a specified point to drop additional ordnance.

Most ASW search tactics are classified. In order to keep this research unclassified, the tactics used were very general in nature.

B. SOURCES OF INFORMATION

As mentioned in Chapter III, the knowledge base for expert systems may be derived from numerous sources, however, most in-depth knowledge comes from the experts themselves. Flying skills and ASW tactical expertise are developed through years of hands on experience. Many aircraft maneuvers and tactical decisions are based purely on *seat of the pants* flying or *this worked last time, so let's try it again*. Much of the knowledge built into the expert system for this study is based on those principles. The author has logged over 3000 hours of piloting experience and hundreds of hours of ASW prosecution time. Many techniques are implemented into the system, not because the reference material says to do it, but because experience has shown that it works.

C. SELECTION OF CLIPS

CLIPS was chosen as the inference engine for this system for a variety of reasons. First, the ASW scenario can be viewed as a forward chaining problem. The crew receives large amounts of information from many sources while on station. They are trained to respond to that information in a specific way. From the crew's viewpoint, ASW is a data driven problem. They receive data and they respond to it. Since CLIPS is designed for forward chaining, it is a likely choice for the ASW problem. Additionally, CLIPS can be easily and effectively interfaced with C, the primary language for graphics software. Finally, the price was right. CLIPS is produced by a government agency (NASA) and is therefore available to other government agencies at no charge.

D. CLIPS SHELL PHILOSOPHY

There is a variety of ways in which to build an expert system with CLIPS. CLIPS can act as the outer shell and control the C program or the C program can act as the outer shell and call CLIPS subroutines for decision making. A third method includes a hybrid of the two previous methods. [NASA91]

Several factors must be considered when designing an expert system with CLIPS:

a. CLIPS is an interpretive language and therefore operates more slowly than normal C code. The larger the number of rules, the slower the code runs.

b. CLIPS is not as efficient at mathematical computations as C. As much of the number processing as possible should be kept in C.

c. CLIPS is very limited in its data structures. The only ones allowed are individual values (i.e. integers, floats, characters, and symbols) and multivalues which is a list of individual values.

For this research, we decided to use CLIPS as the outer shell, driving the C software. The philosophy was to allow the expert shell to serve the function of the mission commander, overseeing the functions of the crew members. The major decisions were made in CLIPS, while the busy work functions of the flight crew were written in C. This maintained a good balance between keeping the code as fast as possible while also taking advantage of the decision making capabilities of CLIPS. Functions for basic aircraft flight maneuvers, buoy monitoring, and target position computation were written in C.

E. LOWER LEVEL FUNCTIONS

1. Basic Aircraft Maneuvering Functions

In developing a set of basic aircraft maneuvers written in C, a building block approach was taken, Figure 14. The lowest level contains the most basic function, rolling

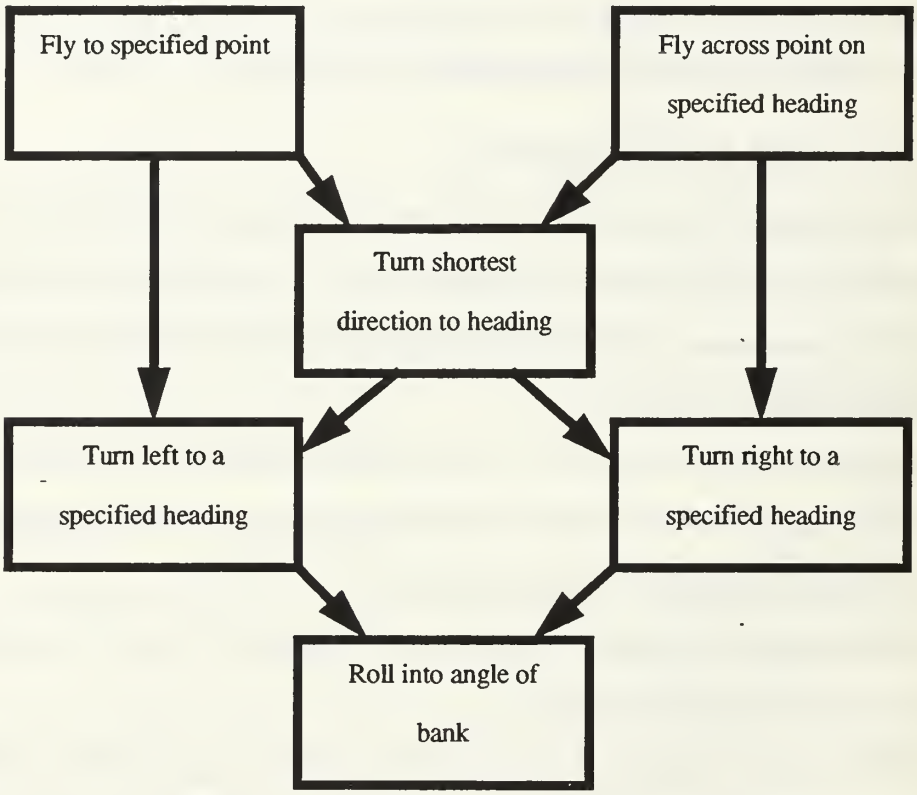
Fly to specified point
Fly across point on specified heading
Turn shortest direction to heading
Turn left to a specified heading
Turn right to a specified heading
Roll into angle of bank

Figure 14 - Aircraft Maneuver Hierarchy

into a desired angle of bank. Each successive level uses lower levels to perform more advanced functions such as rolling into a 45 degree right turn until reaching a desired heading. Many of these maneuvers incorporate the *seat of the pants* flying mentioned earlier. The maneuvers are discussed in more detail below.

a. bank_aircraft

Figure 15 shows the algorithm for banking the aircraft into a desired angle of bank. Positive angles represent right turns and negative angles represent left turns. First the expected angle change during the cycle is calculated based on the time difference between frames and is compared to the difference between the target angle of bank and the current angle of bank. If the current angle of bank is within the expected angle change (i.e. close to the desired angle), then the roll maneuver is complete and the function returns a value of 1. Otherwise, the function increases or decreases the aircraft angle of bank as necessary and returns a value of 0 to indicate the target angle of bank has not yet been reached. In practice, the calling program repeatedly calls bank_aircraft with the desired angle of bank until a 1 is returned.

```
/* Rolls aircraft into specified angle of bank */
/* Positive angle_of_bank is right turn, negative is left turn */
int bank_aircraft(float target_angle_of_bank)
{
  float roll_rate = 10.0; /* roll rate in degrees per second */
  /* delta_time is global for time difference between current and last frame. */
  float angle_change = roll_rate * delta_time;
  /* rx is global for aircraft roll about x axis in degrees */
  float difference = target_angle_of_bank - rx;

    /* If the distance to go is less than normal change, we're done */
    if (fabsf(difference) <= angle_change)
      {
      rx = target_angle_of_bank;
      return(1);
      }
    /* We still need to change AOB, so add or subtract roll rate */
    if (difference >=0.0)
      rx += angle_change;
    else
      rx -= angle_change;
    /* signal that we have not yet reached target angle of bank */
    return(0);
} /* end bank_aircraft */
```

Figure 15 - Algorithm for Banking Aircraft

b. turn_rt_to_heading

This function calls the bank_aircraft function to turn right to a specified heading, using the maximum allowed angle of bank. Turns are forced to the right, regardless of distance remaining. Figure 16 shows the algorithm. When turning the aircraft,

```
/* Turns the aircraft right to the specified heading */
int turn_rt_to_heading(float target_heading)
{
  /* ry is global for aircraft rotation around y (yaw) axis */
  /* Subtracting from 360 aligns rotation with compass headings */
  float max_aob = 50.0; /* maximum angle of bank */
  float ac_heading = 360.0 - ry;
  float delta_heading;
    /* Compute number of degrees to go to target heading */
    delta_heading = target_heading - ac_heading;
    /* Convert heading to between 0 and 360 degrees */
    if (delta_heading < 0) delta_heading += 360.0;
      /* Heading is within 1 degree, zero it out and level wings */
    if (delta_heading <= max_aob/3.0)
       /* If less than 1/3 angle of bank to go, set AOB to 1/3 of remainder */
       bank_aircraft(delta_heading * 3.0);
    else
      bank_aircraft(max_aob);
    /* If we have reached the heading, return a 1 (true) */
    if (delta_heading < 0.15)
  -   return(1);
    else
 /* Signal that we haven't reached the heading yet */
      return(0);
} /* end turn_rt_to_heading */
```

Figure 16 - Algorithm for Turning Right to a Specified Heading

a general rule of thumb used by pilots is to turn at the maximum angle of bank until within one third the angle of bank of the desired heading. Then the pilot starts rolling out of his turn. For example, if turning at a 45 degree angle of bank, when the aircraft gets within 15 degrees of the desired heading, the pilot starts to roll out. This algorithm incorporates the rule of thumb by setting the angle of bank equal to three times the distance remaining or maximum bank angle, whichever is less. The end result is a smooth roll out to wings level on the desired heading. Positive rotation around the y axis is in the opposite direction as the heading markers on the compass. To compensate for this, the headings are subtracted from 360 during calculations. If the desired heading has been reached, the function returns a 1, otherwise it returns a 0.

c. *turn_left_to_heading*

This function performs in the same manner as turn_rt_to_heading except the turn is forced to the left instead of the right. Figure 17 displays the algorithm.

```
/* Turns the aircraft left to the specified heading */
int turn_left_to_heading(float target_heading)
{
  /* ry is global for aircraft rotation around y (yaw) axis */
  /* Subtracting from 360 aligns rotation with compass headings */
  float max_aob = 50.0; /* maximum angle of bank */
  float ac_heading = 360.0 - ry;
  float delta_heading;
    /* Compute number of degrees to go to target heading */
    delta_heading = ac_heading - target_heading;
    /* Convert heading to between 0 and 360 degrees */
    if (delta_heading < 0) delta_heading += 360.0;
      /* Heading is getting close, start rolling out*/
    if (delta_heading <= max_aob/3.0)
       /* If less than 1/3 angle of bank to go, set AOB to 1/3 of remainder */
       bank_aircraft(-delta_heading * 3.0);
    else
      bank_aircraft(-max_aob);
    /* If we have reached the heading, return a 1 (true) */
    if (delta_heading < 0.15)
      return(1);
    else
 /* Signal that we haven't reached the heading yet */
      return(0);
} /* end turn_left_to_heading */
```

Figure 17 - Algorithm for Turning Left to a Specified Heading

d. turn_shortest_heading

This function determines the shortest number of degrees to reach the desired heading and then calls *turn_rt_to_heading* or *turn_left_to_heading* as appropriate until reaching the desired heading. Figure 18 displays the algorithm.

```
/* Picks shortest path from current heading to target heading */
int turn_shortest_heading(float target_heading)
{
 float ac_heading = 360.0 - ry;
 float delta_heading;
 delta_heading = target_heading - ac_heading;
 if (delta_heading < 0) delta_heading += 360.0;
if (delta_heading <= 180.0)
    /* If less than 180 turn right*/
    return (turn_rt_to_heading(target_heading));
 else
    /* Greater than 180, turn left */
    return (turn_left_to_heading(target_heading));
} /* end turn_shortest_heading */
```

Figure 18 - Algorithm for Turning Shortest Distance to Specified Heading

2. fly_to_point

Maneuvering the aircraft to fly across a certain point is much more complex than one might think. Before maneuvering the aircraft, the pilot must look at the entire picture. He not only needs to determine where the aircraft is located in relation to the point, but he also needs to consider aircraft heading and speed. If the point is far enough away, he simply turns toward the point. However, if the point is very close and not directly in front of him, it might be physically impossible to turn the aircraft directly to the point. He must turn the aircraft away from the point and increase the distance before turning inbound. The distances required before turning inbound are based on pilot experience, aircraft speed, and the type of aircraft. The distances used in this study are based on 200 knots, the average maneuvering speed of the P-3 while on station.

To develop a set of rules to correctly model this behavior, we determined that at any given point, the aircraft was in one of four states, Figure 19. State 0 is an analysis and

STATE 0: Analysis and Preposition of Aircraft

STATE 1: Turning shortest direction to reach Point

STATE 2: Must turn LEFT to reach Point

STATE 3: Must turn RIGHT to reach Point

Figure 19 - States for Flying Aircraft to a Point

maneuvering state. In this state, we determine the position of the point relative to the aircraft and turn the aircraft away from the point if necessary. The aircraft then enters another state. State 1 means that the aircraft is turning the shortest direction toward the point. It is in a position such that a direct turn is possible. State 2 requires a left turn to the point. State 3 requires a right turn to the point.Figure 20 illustrates an example of how a left turn is required to reach a point that is on the right side of the aircraft. If the aircraft turns right, the radius of turn exceeds the distance to the point. In turning left, the center point of the radius of turn is offset to the left and the aircraft is able to reach the point. In states 2 and 3, when the amount of turn has exceeded 180 degrees, the aircraft enters state 1 and is now able to continue to turn the shortest direction to the point. The conversion to state 1 is required. Otherwise, a slight overshoot of the point will cause the aircraft to continue around another 360 degrees instead of correcting back in the opposite direction.

To aid in determining the point's relative position to the aircraft, we divided the compass rose around the aircraft into eight 45 degree sectors, Figure 21. Using sector numbers categorizes the relative bearing into a small number of workable groups. The current sector number is computed as follows:

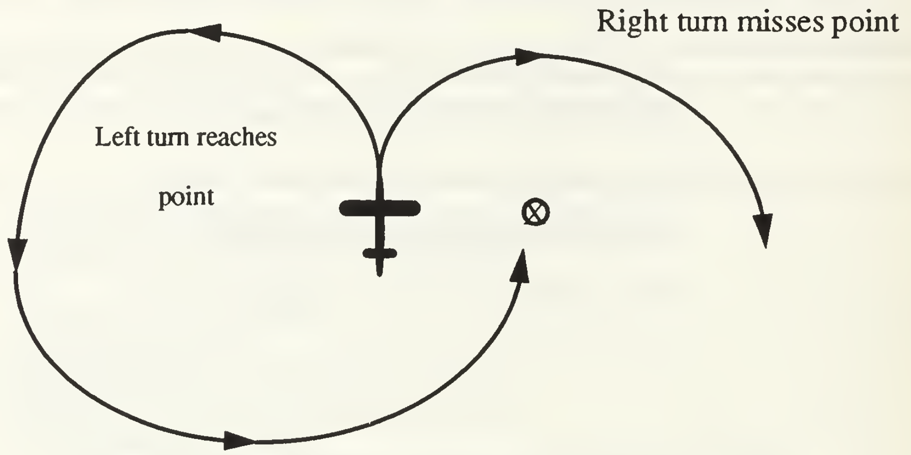

Figure 20 - Left Turn Required to Reach Point on Right

$$sectornumber = floor((relativebearing)/45.0) \qquad \text{(Eq 19)}$$

Each sector has a specific set of rules for the aircraft to follow. In several cases, adjacent sectors have the same set of rules. A description of the sector rules follows.

a. Sectors 0 and 1

In this sector, the point is in front of the aircraft wings and to the right of the nose. We must determine whether the aircraft can turn directly to the point or must turn away to increase the distance. If the point lies within an intercept window to the right of the nose of the aircraft, then the aircraft can turn directly to the point, otherwise it assumes state 2 and turns away from the point to the left, similar that shown in Figure 20. The angular width of the intercept window is directly proportional to the distance to the point and indirectly proportional to the speed of the aircraft. Experimentation produced a window intercept function as follows:

$$windowwidth = \frac{100.0 \times distance}{airspeed} \qquad \text{(Eq 20)}$$

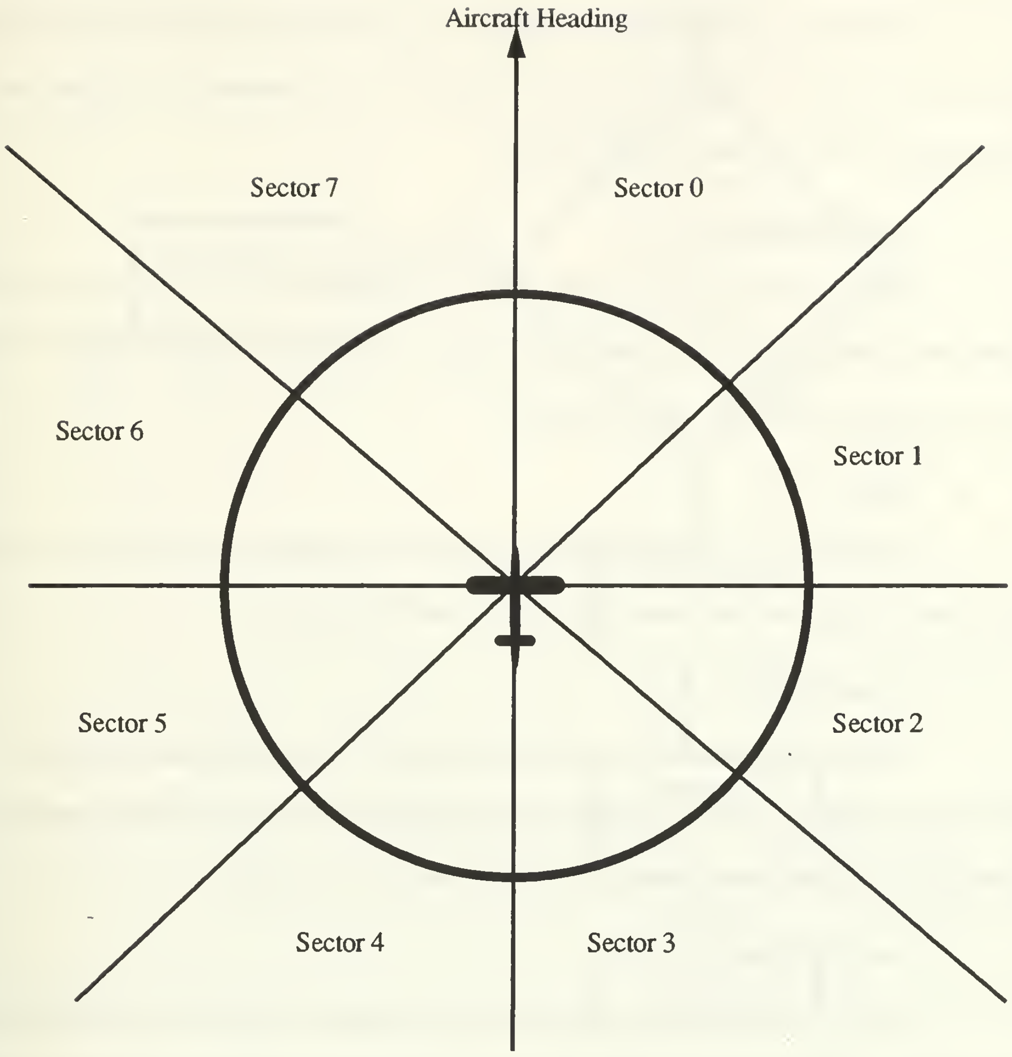

Figure 21 - Relative Sector Assignments for Fly-to-Point

Where:

windowwidth = width of the intercept window in degrees.

distance = distance from point to aircraft in nautical miles.

airspeed = airspeed of aircraft in knots.

b. *Sector 2*

This sector is just behind the right wing. If the point is farther than 2 nautical miles from the aircraft, it turns directly to the point. If the distance is less than 2 nautical miles, the aircraft goes to state 2 and turns left.

c. *Sectors 3 and 4*

These sectors are directly behind the aircraft. In this case, the aircraft maintains its heading until the distance to the point exceeds 2 nautical miles. Then the aircraft turns the shortest distance to the point.

d. *Sector 5*

This sector is behind the left wing. The procedures are similar to sector 2 except the turn away from the point is in state 3 and to the right.

e. *Sectors 6 and 7*

In sectors 6 and 7, the point is in front of the wings and to the left of the aircraft. The procedures for these sectors are very similar to those in sectors 0 and 1 except when turning away from the point, the turn is in state 3 to the right.

f. *Function Flow*

Figure 22 illustrates the overall block diagram for the fly_to_point function. When entering the function, a check is made to see if we have reached the point during this frame. If so, the function returns a 1 and exits. If not, the sector and range are determined and from that the proper state is set and the new aircraft position and orientation are computed. A zero is returned to signal that the aircraft has now yet reached the point. Appendix A contains the C source code for the fly_to_point function.

3. designated_fly_to_point

The designated_fly_to_point function flies the aircraft over a point on a specified heading. This function is useful when dropping a row of sonobuoys or when attacking a submarine down its course. For example, when dropping a row of buoys oriented from east

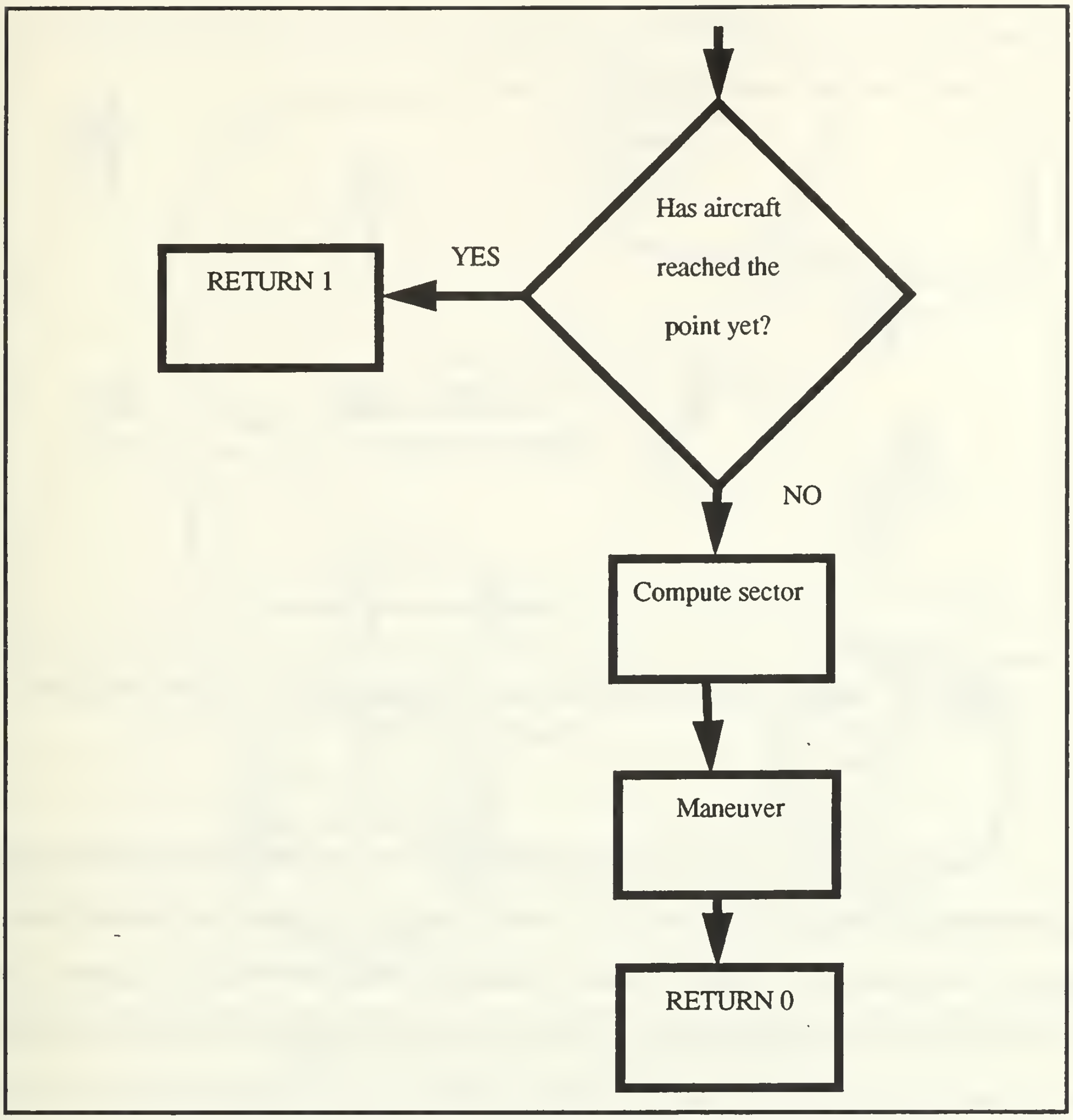

Figure 22 - Block Diagram for fly_to_point Function

to west, if the aircraft crosses the first buoy in the row on a heading of west, it will already be lined up to drop the remaining buoys in the row without excessive maneuvering. Figure 23 illustrates this example. To get the aircraft in position, it must be maneuvered to the same side of the point as the inbound heading. A typical maneuver is to fly toward the point to decrease the distance, and then to fly outbound on a heading opposite the inbound heading. When sufficiently down course, the aircraft turns to intercept the inbound course. Aircraft A and B in Figure 23 show two examples of this maneuver.

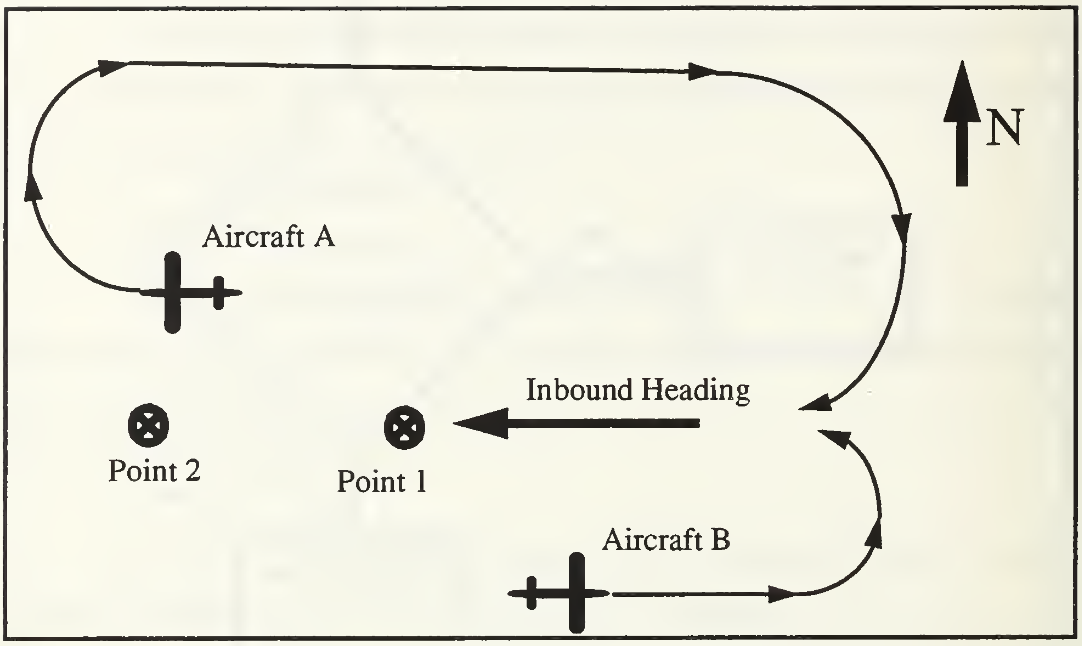

Figure 23 - Aircraft Positioning During Designated Fly To Point Function

A technique that is taught in pilot training, to intercept the course is the *double the angle* intercept. Figure 24 illustrates the concept. Aircraft A is abeam the desired point. His compass card shows the aircraft heading straight ahead, the desired intercept course directly behind him, and the relative bearing to the point directly off his left wing. As the aircraft moves forward, the relative bearing pointer will continue to point at the desired point and rotate counterclockwise as the point goes behind the wing. The pilot maintains constant heading until the relative bearing pointer falls to about 45 degrees behind his right wing. At that point, he turns left 90 degrees and is in the position of Aircraft B. At this point, the heading is now pointing north and the inbound course is still pointing west. The relative bearing to the point is now northwest, half way between the other two pointers. Note that the aircraft heading is at twice the angle between the inbound course and the relative bearing, hence the name *double the angle*. The pilot now continues to turn slowly left, keeping the relative bearing half way between the two other two pointers. Aircraft C is in

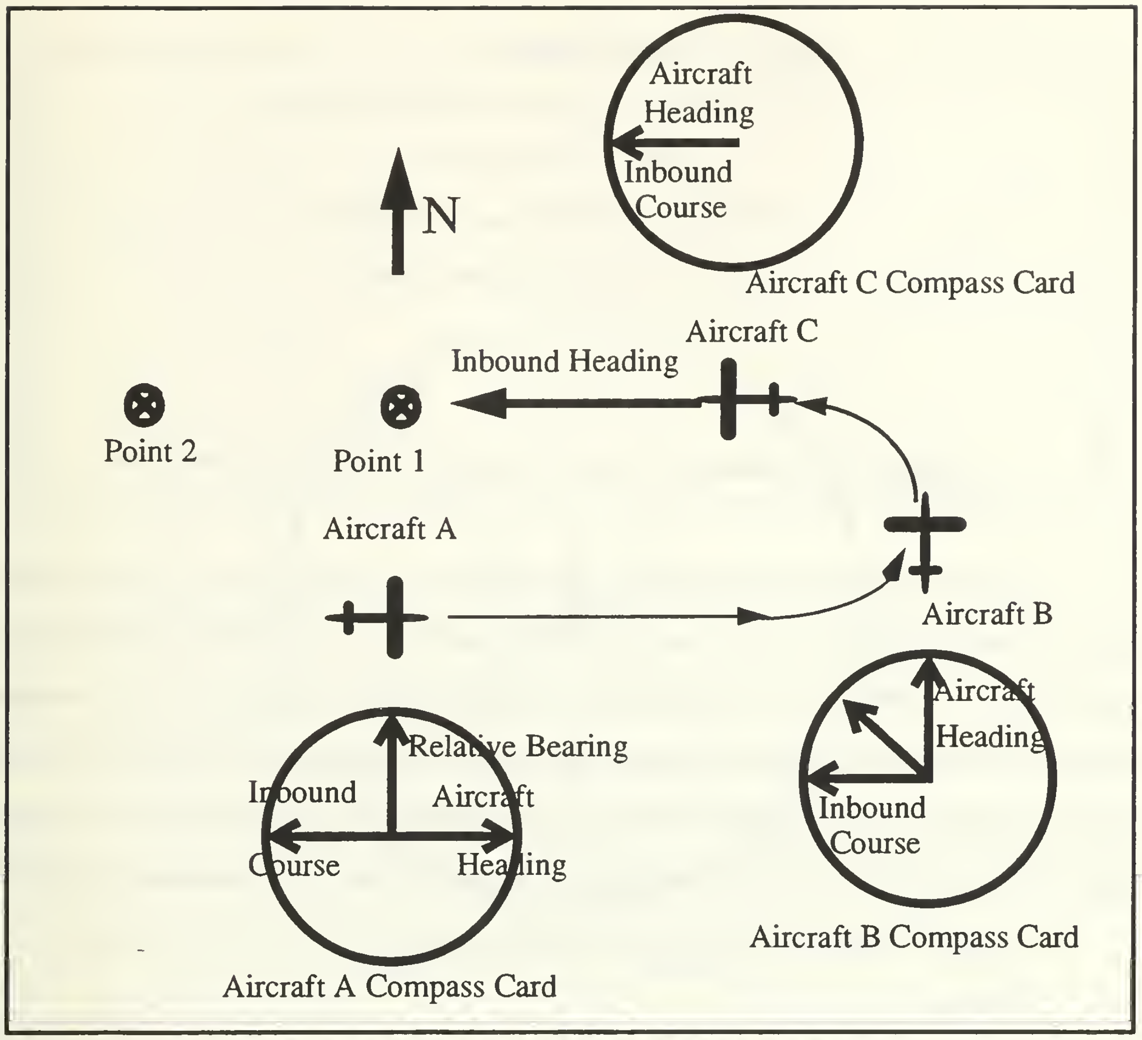

Figure 24 - Double the Angle Intercept

the final stages of the maneuver. The aircraft eventually ends up on the desired inbound heading with all three pointers pointing toward the inbound heading.

The states for this function are different from those in the *fly_to_point* function. State 0 checks the aircraft distance from the point and turns the aircraft toward the point until it reaches the transition distance which is two nautical miles. State 1 determines the correct sector and transitions to State 2 or 3. State 2 indicates the aircraft is in the correct position to start the double angle intercept. State 3 is when the aircraft is turning down course in preparation for the double angle intercept. Figure 19 summarizes the states.

STATE 0: Preposition Aircraft to Transition Range

STATE 1: Determine Sector

STATE 2: Do Double Angle Intercept

STATE 3: Move Aircraft Down Course

Figure 25 - States for Flying Aircraft to a Designated Point

The sectors are also computed a little differently from the *fly_to_point* function. Instead of centering the sectors around the aircraft, the sectors are centered around the point and the sector is determined by the aircraft's position relative to the point. The boundary between sector 7 and sector 0 is the reciprocal of the inbound heading. Figure 26 illustrates the sector layout. The orientation based on point position is essential since the inbound heading is relative to the buoy, not the aircraft. The sector number is computed as follows:

$$sector = floor\left((truebearing - desheading)/45.0\right) \qquad \text{(Eq 21)}$$

Where:

sector = the sector number.

truebearing = true bearing from the point to the aircraft.

desheading = desired input heading.

Note: The quantity (*truebearing* - *desheading*) is normalized between 0 and 360 degrees before division by 45.0.

The procedures in the sectors fall into to basic groups.

a. *Sectors 0 and 7*

The aircraft is on the inbound heading side of the point. A check is made to see if the point is within the aircraft's intercept window. If so, the aircraft enters State 2 and

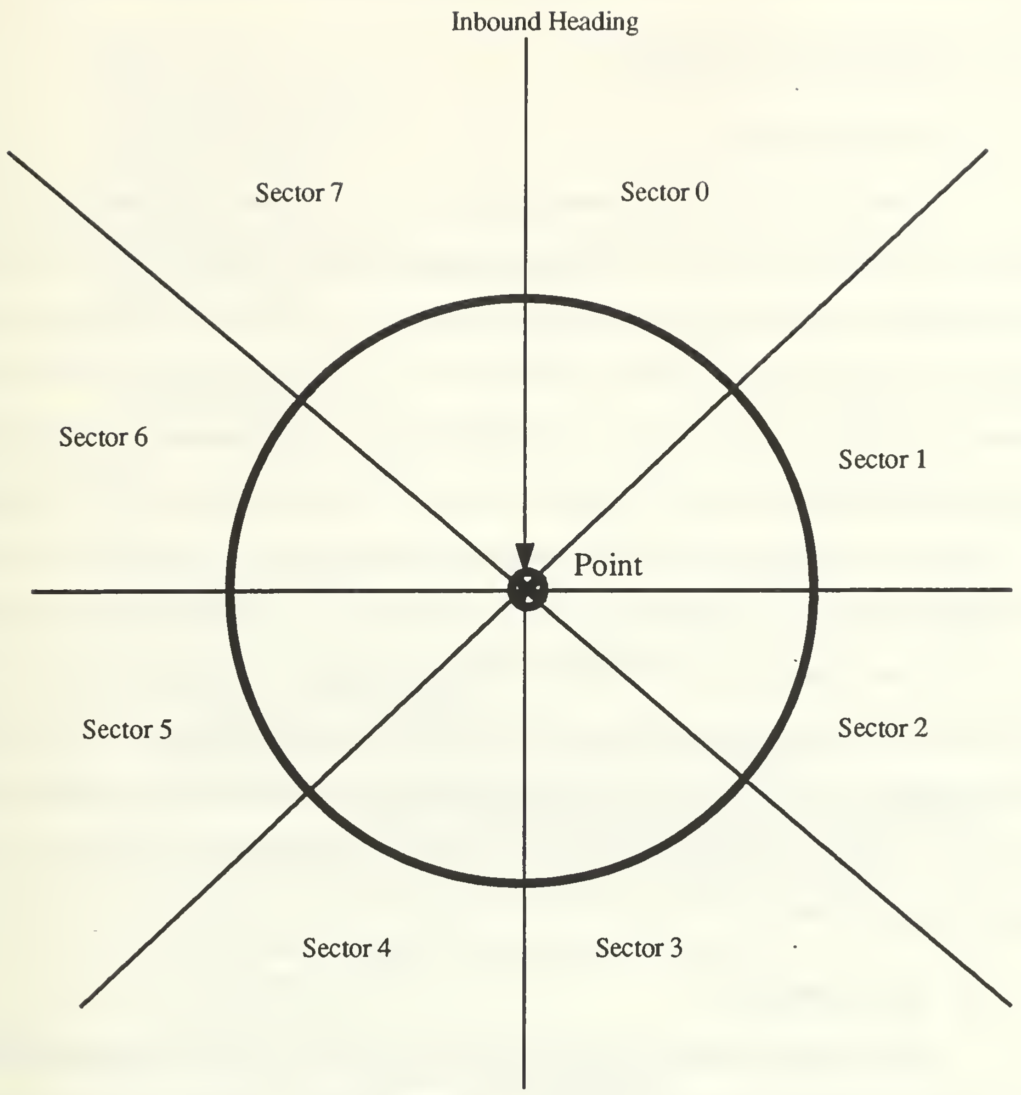

Figure 26 - Sector Assignments for Designated Fly To Point

starts the double angle intercept. If the point is not within the aircraft's intercept window, the aircraft enters state 3 and turns outbound to increase the distance from the point.

b. States 1 through 6

In these states, the aircraft is not in a position to start the double angle intercept. The aircraft is turned opposite the inbound course and proceeds until the relative bearing is 45 degrees below one of the wingtips.

The code for the *designated_fly_to_point* function is listed in Appendix B.

4. Miscellaneous C Functions

a. monitor_buoys

monitor_buoys is a function which serves the function of the sensor operators. It simply scans all buoys that are in the water and stores the buoy number, buoy position and true bearing to the target in a contact data array. All buoys for this research were considered to be ideal buoys. That is, if the buoy is in the water, it has contact on the submarine. In reality, for a sonobuoy to develop contact on a submarine, it must overcome a large number of parameters such as signal strength of the submarine, ambient background noise, distance from the buoy to the submarine and sensitivity of the buoy. To avoid overloading the expert system with data from ideal buoys, *monitor_buoys* maintains an array of the best two buoys in contact. During the scan of the buoys for contact, *monitor_buoys* also computes the distance from the buoy to the submarine. If that distance is less than either of the two buoys in the best buoy array, the new buoy replaces the best buoy that is farthest from the submarine. Figure 27 shows the algorithm for the *monitor_buoys* function.

b. find_target_posit

find_target_posit is a function which determines the target location given two buoys in contact. The function creates two line segments in the X-Z plane, extending from the buoy positions. It then uses a line intersection function written by Prasad [Arvo91] to determine the submarine's location.

F. CLIPS INTERFACE WITH GRAPHICS ENVIRONMENT

The process of interfacing functions and passing data between CLIPS and C is a straightforward, yet tedious procedure. The mechanics of the interface include defining a series of functions, written in C, which can be called by CLIPS. The *Clips Reference Manuals* [NASA91] provide in-depth details and examples on the mechanics of building

```
void monitor_buoys()
{
  int i, j, k;
  float b_pos[3], sub_pos[3];
  float contact_tx, contact_tz;
  sub_pos[0] = sub_tx;
  sub_pos[1] = sub_ty;
  sub_pos[2] = sub_tz;
 b_pos[Y] = 0.0;
 /* Initialize best_2_buoys to maximum range*/
  for (j=0; j<2; j++)
   {
   for (k=0; k<5; k++)
     best_2_buoys[j][k] = 0.0;
    best_2_buoys[j][RANGE] = 10000.0;
   }
 for (i=0; i<MAX_BUOYS; i++)
   {
   /* If the buoy is in the water, compute the data */
     if (buoy_in_water[i])
      {
        /* extract buoy position out of contact_data array */
        b_pos[X] = contact_data[i][0];
        b_pos[Z] = contact_data[i][1];
        /* Compute the bearing and range from buoy to submarine */
        compute_xz_bearing_dist(b_pos,sub_pos,
                       &contact_data[i][BEARING], &contact_data[i][RANGE]);
        /* See if this buoy is one of the best two buoys in contact */
        if (contact_data[i][RANGE] < best_2_buoys[0][RANGE])
          {
          /* New value is smallest, move it in first slot and first
          slot into second slot */
          for (k=0; k<4; k++)
            {
            best_2_buoys[1][k] = best_2_buoys[0][k];
            best_2_buoys[0][k] = contact_data[i][k];
            }
          /* move the buoy numbers into slots */
          best_2_buoys[1][4] = best_2_buoys[0][4];
          best_2_buoys[0][4] = (float) i;
          }
        else
          if (contact_data[i][RANGE] < best_2_buoys[1][RANGE])
            /* New value is smaller than second slot but larger
                than first slot, move it into second slot.*/
           {
            for (k=0; k<4; k++)
            best_2_buoys[1][k] = contact_data[i][k];
```

Figure 27 - Algorithm for monitor_buoys Function

the interface. However, the manuals do not discuss the peculiarities of interfacing with a graphics environment. In a standard IRIS graphics program, after initialization, the software enters an infinite loop where it sequentially computes the data for the next frame, sends that data down the graphics pipeline and then returns to the beginning of the loop. With the outer CLIPS shell controlling program flow, it is not possible to set up an infinite loop in C to do the drawing. The solution is to place the code contained in the infinite graphics loop into a separate function. The CLIPS rules are written so that they create an infinite loop and call the new function repeatedly.

Figure 28 illustrates the overall block diagram of the CLIPS program flow. In

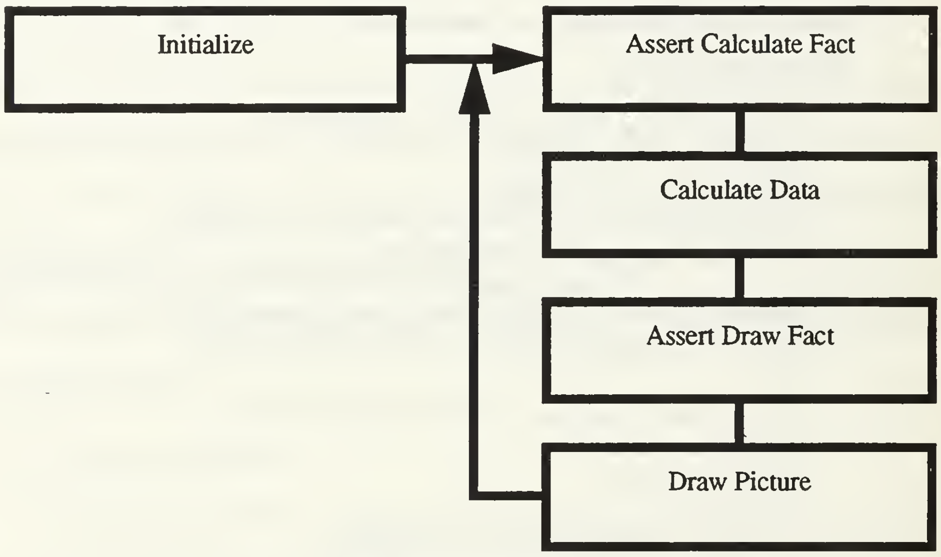

Figure 28 - Block Diagram of CLIPS Outer Shell Program Flow

this system, we divided the rules into three categories: initialization, calculation and drawing. By placing the rules in groups, we allow the rules to fire independently within each group, while maintaining control over when each group of rules can fire. Using this technique, we force all calculating rules to be completed before the drawing rules are enabled. To signal each group of rules that it is safe to start firing, we assert a control fact which acts as an enabling fact for all rules in that group. One rule is provided for each group with a lower priority, or salience, than the other rules in the group. [NASA91] The purpose of the low priority rule is to assert the control fact for the next group of rules. The lower salience guarantees that the rule will fire only after all other rules in the group have finished firing. The end result is an infinite loop with groups of rules being enabled in sequence.

G. DESCRIPTION OF CLIPS SHELL

For readers unfamiliar with CLIPS syntax, CLIPS rules are broken into two parts, the Left Hand Side (LHS) and the Right Hand Side (RHS). The two parts are separated by the => symbol. The rule is analogous to a large if..then statement. The facts on the LHS are compared to the current fact list. If all of the facts on the LHS are satisfied, then all of the functions on the RHS are executed Facts may be asserted onto or retracted from the fact list. See [Giar91] and [NASA91] for a more detailed description of the CLIPS language.

Appendix C lists the facts and rules for the CLIPS shell used in this study. To prove the concept of an expert systems controlled P-3, we chose a simple ASW scenario which drops a pre-briefed sonobuoy pattern, monitors the buoy pattern, tracks the submarine and, when given authorization to attack, flies to the submarine and drops a torpedo on it. After the attack, the P-3 flies to a designated point and orbits in the area.

1. Initialization rules

To initialize the system, the CLIPS shell accomplishes two things. First, it initializes the graphics environment. Then, it reads the briefed buoy pattern file, *waypoints.dat*. The rule *init-p3*[1] looks for (initial-fact) and calls the graphics initialization function *init_off_objects*. It also asserts (read-file) which signals the file reading rules to read the buoy data file.

Three rules work together to read the buoy data file, *read-input-file, read-file*, and *close-all-files. read-input-file* reads the next line in the file and asserts the data onto the fact

1. For convention, rule names are italicized and fact names are enclosed in parenthesis.

list. *read-file* pulls the data off of the fact list and splits it into buoy number, buoy position and input heading for drop.

2. Computation Rules

The computation rules can be divided into several functional areas: reading boolean flags from C, dropping the sonobuoy pattern, monitoring the pattern, attacking the submarine, orbiting at a specified point, and performing miscellaneous housekeeping functions. The rules in this section fire in a sequence determined by the order of the facts on the fact list and by the CLIPS language environment.

The rule *get-script-flag* reads in the boolean flags required by CLIPS and assigns their values to a set of global boolean variables. After reading the flags, it asserts (continue-computing) to enable the remaining computation rules.

Six rules work together to drop the sonobuoy pattern in the correct sequence. *continue-prosecution* checks the prosecution flag which is sent by the C code. It asserts (start-drops) to start the buoy pattern. *keep-dropping* finds the next buoy number to be dropped and puts it on the fact list. *pattern-complete* fires if the last buoy number has been dropped and asserts (pattern-complete) to stop the buoy drops. *drop-buoys* pulls the next buoy off of the fact list and directs the aircraft to go to that point. *not-reached-buoy-drop* fires if the aircraft has not yet reached the point. It signals to continue to point. *reached-buoy-drop* fires if the aircraft has reached the point. The rule drops a buoy and asserts (get-next-buoy) to signal other rules to pull the next buoy off of the fact list.

Three rules work together to control the aircraft after the pattern has been dropped. *after-pattern-orbit* sends the aircraft to a point near the pattern and keeps directing the aircraft to fly to that point. The result is a figure 8 pattern over the point. *monitor-pattern* collects data on the strongest two buoys in contact. *get-fix* computes the submarine position based on data from the best two buoys.

Four rules work together to drop the torpedo on the target. *attack* looks for attack authorization from C and asserts (attack-submarine) to start the attack. *attack1* directs the aircraft to fly to the submarine location. *not-reached-torp-drop-point* fires if the aircraft has not yet reached the target. *reached-drop-point* fires when the aircraft reaches the point. It executes the drop torpedo command to C and asserts (finished-attacking) to enable rules to direct the aircraft to the orbit point.

The rule *end-compute-sequence* has been declared with a salience of -10. This ensures that all other computation rules are finished firing before it fires. It asserts the fact (ok-to-draw) which signals the drawing rules to fire.

3. Graphics Drawing Rules

There is only one rule, *main-p3*, used to execute the graphics drawing command. If the (ok-to-draw) fact has been asserted, the rule *main-p3* fires. It calls the main graphics program, *p3_main*, once and then asserts (ok-to-compute) to enable the computational rules and start the cycle over again.

VII. NETWORKING THE MODEL

A. BACKGROUND

Networking has become a very significant part of simulation systems. The large size and complexity of simulation scenarios today are stressing the limits of even the most powerful computing machines. To overcome this limitation, it has become common practice to break large simulation scenarios into smaller stand-alone simulation systems which interact with each other over a high speed network. These interactive units can be in the same building or on the other side of the world. With this in mind, we designed this system to be a totally separate unit on its own host machine, computing aircraft positional data and broadcasting that data over a network. We used the Distributed Interactive Simulation (DIS) [IST91] protocol to format our packets and transmit them over ethernet to other graphics stations in the NPS graphics lab.

B. DIS PROTOCOL

A relatively new simulation protocol is the DIS protocol which is designed to replace the SIMNET [Pope89] protocol. DIS is a highly structured system which has the capability to send extensive information about units over a network. The packet used to send the data is known as a Protocol Data Unit (PDU). DIS version 1.0 has ten types of PDUs:

Entity State PDU.
Fire PDU.
Detonation PDU.
Service Request PDU.
Resupply Offer PDU.
Resupply Received PDU.
Resupply Cancel PDU.
Repair Complete PDU.
Repair Response PDU.
Collision PDU.[IST91]

PDUs used for this study are the Entity State PDU, the Fire PDU, and the Detonation PDU. The Entity State PDU is used to communicate information about a unit's current state, including position, orientation, velocity, and appearance. The Fire PDU contains data on any weapons or ordnance that is fired. In this work, the Fire PDU is used to transmit sonobuoy and torpedo drops. The Detonation PDU is sent when a piece of ordnance impacts the ground and detonates. For this work, the detonation PDU is used to transmit buoys hitting the water and torpedoes exploding.

The actual structure of a PDU is very regimented and is explained in full detail in [IST91]. The software we used to format the packets and send them over the network was produced in house by Mr. John Locke. [Lock]

C. PLAYERS AND GHOSTS

The networking technique we used follows the *players and ghosts* paradigm presented in [Blau92]. In this paradigm, every real world object is controlled on its home machine by a software object called a Player. On every other machine in the network that will be displaying the object, a dead reckoning version of the Player, called the Ghost, must be present. The primary purpose of the Ghost is to prevent network overload by using dead reckoning. If every system tried to send out an updated packet on each of its objects after every frame, the net would be saturated with packets and the system would quickly bog down. With Players and Ghosts, the Player only sends out a packet whenever a significant change occurs, such as a heading change or an ordnance drop. Between packets, the Ghost computes the next position or attitude of the Player using dead reckoning. When a new packet is received, the Ghost compares the new data with the dead reckoned data and adjusts it accordingly. This paradigm allows accurate data to be displayed at each machine without saturating the network.

D. IMPLEMENTING DIS IN THE SYSTEM

1. Sending P-3 Entity State PDUs

The Entity State PDU is one of the largest PDUs in DIS and contains a lot of information that remains constant for a given object, such as protocol version, exercise ID, country, category, etc. The only items that routinely change are position, orientation, velocity and time of the PDU. To save computer time, we constructed a PDU initialization routine which set all of the constant PDU values. When transmitting a new PDU, we simply updated the time, position, attitude and velocity values and then transmitted the packet.

[IST91] indicates that a primary method for determining when to send an Entity State PDU is to track both the actual position and the dead reckoning position on the host machine and send a PDU when they differ by a certain amount. An alternative method is to send a PDU when a predetermined period of time has elapsed. We chose a method based on the latter which counts frames since the last PDU was sent. Through experimentation we determined that sending a PDU every 15 frames produced a smooth image at the receiving machine without saturating the net. The advantage of this method is that time is not wasted computing an extra dead reckoning position.

2. Receiving P-3 Entity State PDUs

When an Entity State PDU for a P-3 is received, the current position is computed by subtracting current system time from the time of transmission and applying Equations (Eq 1), (Eq 2), (Eq 3), and (Eq 4). On subsequent frames, the new dead reckoning position is computed based on the same equations.This conforms to first order positional dead reckoning or Formula 1 in [IST91].

3. Sending and Receiving Sonobuoy Entity State PDUs

When dropping a sonobuoy, a Fire PDU is transmitted, a new sonobuoy object is created and a series of Entity State PDUs is sent as the sonobuoy falls. Sending Entity State PDUs for a sonobuoy is similar to sending a P-3 Entity State PDU, except the opening angle of the stabilizing fins must be included. This is done by placing the value in the *change* slot of the *articulated parameters node* in the Entity State PDU. When the sonobuoy hits the water, a Detonate PDU is transmitted and transmission of Entity State PDUs for the sonobuoy halts.

When receiving sonobuoy Entity State PDUs, the position of the sonobuoys is dead reckoned using the sonobuoy equations in Chapter IV, until the buoy hits the water.

The physical location of the buoy after entering the water is only used by the host unit and further updates on the net are not necessary.

4. Sending and Receiving Torpedo Entity State PDUs

The procedures for sending and receiving PDUs for torpedo drops is virtually identical to those used for sonobuoy drops, except that there is no need to provide stabilization fin information.

E. TESTING THE CONCEPT

To test the concept of using DIS to network data for the P-3, we embedded both the Player and the Ghost software for the aircraft, sonobuoys and torpedoes in the same program. We ran the program simultaneously on two machines, machine A and machine B, with networking enabled. Each machine displayed two separate aircraft. One aircraft was being generated by machine A while the other was being generated machine B. Each aircraft was independently controlled by its host machine and changes in attitude and position were updated in real time on both machines.

VIII. CONCLUSIONS AND FUTURE WORK

A. CONCLUSIONS

The primary purpose of this work was to prove the concept of controlling a P-3 aircraft in an ASW environment with an expert system and networking the results to NPSNET. Sub-areas of study included building the graphical and flight dynamics models of the P-3 and modeling the flight dynamics of the ordnance dropped by the aircraft.

After performing the development, testing and evaluation of the various features of this project, we have reached the following conclusions:

- A realistic, transportable, graphical model of the P-3 aircraft can be produced.
- A simplified, yet realistic flight dynamics simulation is feasible.
- The ordnance dropped by the aircraft can be realistically modeled.
- Networking data to NPSNET using DIS protocol is feasible. The latest version of NPSNET which incorporates DIS is in production and this work can be easily imbedded in it.
- Most importantly, all of the above features can be done real-time.

B. FUTURE WORK

Since this work is the first to use maritime aircraft and an expert system with NPSNET, there is an unlimited number of areas where it can be expanded with future studies. Some of these include:

- A more precise aerodynamic model of the P-3 could be included. Previous work [Cook92] has already laid the groundwork for this.
- A submarine controlled by an expert system could be implemented. The submarine model used for this study maneuvered in the area on a scripted track. An expert system submarine could be programmed to react to the P-3 when it flies over or when a torpedo is dropped.
- The sonobuoy behavior in the water could be made more realistic. Adding physical models of underwater acoustic properties would enhance the ASW scenario, making it more true to life.
- Surface ships driven by expert system could be added to the project. This expansion would significantly enhance the simulation of

coordinated operations between ships and aircraft in an ASW environment. Work in this area is already underway at NPS.

- Expand the system to include sonobuoy pattern expansion tactics when only one buoy has contact.

The following enhancements are possible, but are not recommended at this time because of their classification level. The NPS graphics lab does not currently have the equipment nor facilities to handle classified work.

- Develop an expert system to model the behavior of the MK-46 torpedo when it enters the water.
- Expand the search and tracking tactics used by the aircraft to more closely match those actually used by P-3 flight crews.

This work is a first step in incorporating naval units into NPSNET. The possibilities for expansion are limited only by the imaginations of future users.

APPENDIX A. SOURCE CODE FOR fly_to_point

```
/*************************************************************
Function:  fly_to_point(float destination[3])

Description:  Performs the required functions to maneuver the
    aircraft correctly to intercept a specified point.
    The compass rose is divided into 8 equal sectors and the
    algorithm determines which action to perform based on
    sector, range to point and aircraft speed.

Input: destination - xyz coordinates of desired point.

Output: the function returns 1 when it has reached the point, otherwise
   it returns a 0.

*************************************************************/
int fly_to_point(float destination[3])
{
 float true_bearing, range, rel_bearing;
 float current_pos[3];
 int i, sector;
 static  float last_fly_to_point[3] = {
  0.0, 0.0, 0.0  };
 float temp_dist, dummy;

 current_pos[X] = tx;
 current_pos[Y] = ty;
 current_pos[Z] = tz;

 /* See if a new fly to point is being sent. If so, start over! */
 compute_xz_bearing_dist(last_fly_to_point,destination,&dummy,&temp_dist);
 if (temp_dist > 5.0)
 {
  ftp_state_variable = 0;
  printf("New ftp = %f %f\n", destination[X], destination[Z]);
 }
 for (i=0; i<3; i++)
  last_fly_to_point[i] = destination[i];

 /* If we are within range of our destination,
    signal we are done and reset state.  */
 if ((fabsf(destination[0] - current_pos[0]) < 2.0) &&
    (fabsf(destination[2] - current_pos[2]) < 2.0))
 {
  printf("Captured fly-to-point: x = %f z = %f\n",
  destination[0], destination[2]);
  ftp_state_variable = 0;
  return(1);
```

```
}
else
  {
 switch(ftp_state_variable)
 {
 case 0:
  /* Find bearing and range to destination and turn towards it */
  compute_xz_bearing_dist(current_pos, destination, &true_bearing, &range);
  rel_bearing = relative_bearing(360.0-ry, true_bearing);
  /* find sector        - 0  equals   0 to  45 deg
                          1  equals  46 to  90 deg
                          2  equals  91 to 135 deg
                          3  equals 136 to 180 deg
                          4  equals 181 to 225 deg
                          5  equals 226 to 270 deg
                          6  equals 271 to 315 deg
                          7  equals 315 to 359 deg  */
  sector = (int) (rel_bearing / 45.0);

  switch(sector)
  {
  case 0:
  case 1:
   /* if point within intercept window, go direct,
            otherwise turn right */
   if (rel_bearing <= compute_intercept_window(range, velocity))
     ftp_state_variable = 1;
   else
      /* Not in window, go left */
   ftp_state_variable = 2;
   break;
  case 2:
   /* point is on right side, slightly behind. If far enough away
            turn directly toward, otherwise turn left       */
   if (range > 200.0)
     ftp_state_variable = 1;
   else
      ftp_state_variable = 2;
   break;
  case 3:
  case 4:
   /* point is behind aircraft, wait until distance opens to */
   /* 100 and then start turn to shortest direction */
   if (range > 100.0)
     ftp_state_variable = 1;
   break;
  case 5:
   /* point is on left side, slightly behind. If far enough away
            turn directly toward, otherwise turn right       */
   if (range > 200.0)
     ftp_state_variable = 1;
```

```
        else
          ftp_state_variable = 3;
        break;
      case 6:
      case 7:
        /* if point within intercept window, go direct,
                    otherwise turn right   */
        if (360.0 - rel_bearing <= compute_intercept_window(range, velocity))
          ftp_state_variable = 1;
        else
          /* Not in window, go left */
        ftp_state_variable = 3;
        break;
      default:
        ftp_state_variable = 1;
        break;
      } /* end switch(sector) */

    case 1:
      /* Find bearing and range to destination and turn shortest direction */
      compute_xz_bearing_dist(current_pos, destination, &true_bearing, &range);
      turn_shortest_heading(true_bearing);
      break;
     case 2:
      /* Find bearing and range to destination and turn left towards it */
      compute_xz_bearing_dist(current_pos, destination, &true_bearing, &range);
      turn_left_to_heading(true_bearing);
      rel_bearing = relative_bearing(360.0-ry, true_bearing);
      if (relative_bearing(360.0 - ry,true_bearing) > 270.0)
        ftp_state_variable = 1;
      break;
     case 3:
      /* Find bearing and range to destination and turn right towards it */
      compute_xz_bearing_dist(current_pos, destination, &true_bearing, &range);
      turn_rt_to_heading(true_bearing);
      rel_bearing = relative_bearing(360.0-ry, true_bearing);
      if (relative_bearing(360.0 - ry,true_bearing) < 90.0)
        ftp_state_variable = 1;
      break;

     default:
      ftp_state_variable = 0;
      break;
     }  /* end switch */

     /* Signal that we aren't there yet */
     return(0);
   } /* end else */
 }  /* end fly_to_point */
```

APPENDIX B. SOURCE CODE FOR designated_fly_to_point

```
/*****************************************************
Function: designated_fly_to_point(float destination[3], float des_heading)

Description: Performs the required functions to maneuver the
     aircraft correctly to intercept the point (called destination)
     on the input true heading (called des_heading).

Input: destination - xyz coordinates of desired point.
      des_heading - true heading to hit the point

Output: the function returns 1 when it has reached the point, otherwise
    it returns a 0.

*****************************************************/
int designated_fly_to_point(destination, des_heading)
float destination[3], des_heading;
{
 float transition_dist = 200.0;  /* Distance at which to start turn in */
 float true_bearing, range, rel_bearing;
 float current_pos[3];
 float angle_difference;
 int i, sector;
 float double_heading;
 float temp_dist, dummy;

 static float last_des_ftp[3] = {
   0.0, 0.0, 0.0  };
 static float last_heading = 0.0;

 current_pos[X] = tx;
 current_pos[Y] = ty;
 current_pos[Z] = tz;

 /* See if a new fly to point or heading is being sent. If so, start over! */
 compute_xz_bearing_dist(last_des_ftp, destination, &dummy, &temp_dist);
 if (temp_dist > 5.0)
   dftp_state_variable = 0;

 for (i=0; i<3; i++)
   last_des_ftp[i] = destination[i];

 /* If we are within range of our destination,
    signal we are done and reset state.  */
 /* Find the difference between aircraft heading and designated heading */
 angle_difference = normalize_degrees(360.0 - ry - des_heading);
 /* If the difference is over 180.0, sub 180 to get the smallest angle.*/
 if (angle_difference > 180.0)
```

```
  angle_difference -= 360.0;
/* See if we have captured the point */
if ((fabsf(destination[0] - current_pos[0]) < 2.0) &&
    (fabsf(destination[2] - current_pos[2]) < 2.0) &&
    (fabsf(angle_difference) < 5.0))
{
  printf("Captured designated fly-to-point: x = %f z = %f\n",
  destination[0], destination[2]);
  dftp_state_variable = 0;
  return(1);
}
else
   {
  switch(dftp_state_variable)
  {
  case 0:
    /* Find bearing and range to destination and turn towards it */
    compute_xz_bearing_dist(current_pos, destination, &true_bearing, &range);
    rel_bearing = relative_bearing(360.0-ry, true_bearing);
    /* If long distance from point, get closer before maneuvering */
    if (range > transition_dist)
      turn_shortest_heading(true_bearing);
    else
      dftp_state_variable = 1;
    break;
  case 1:       /* Distance to point is less than transition distance */

    /* find sector        - 0  equals   0 to  45 deg
         relative to        1  equals  46 to  90 deg
       designated input     2  equals  91 to 135 deg
         heading            3  equals 136 to 180 deg
                          4  equals 181 to 225 deg
                          5  equals 226 to 270 deg
                          6  equals 271 to 315 deg
                          7  equals 315 to 359 deg  */

    compute_xz_bearing_dist(current_pos, destination, &true_bearing, &range);

    rel_bearing = relative_bearing(360.0-ry, true_bearing);
    sector = (int) (normalize_degrees(true_bearing - des_heading) / 45.0);

    switch(sector)
    {
    case 0:
    case 7:
      /* we are in the front sectors, are we in posit to intercept */
      if (rel_bearing <= compute_intercept_window(range, velocity))
        /* We can make it, start turning inbound */
        dftp_state_variable = 2;
      else
        /* We cannot make it, turn outbound */
```

```
          dftp_state_variable = 3;
        break;
      case 1:
      case 2:
      case 3:
      case 4:
      case 5:
      case 6:
        dftp_state_variable =3;
        break;
      default:
        dftp_state_variable = 1;
        break;
      } /* end switch(sector) */

      break;

    case 2:    /* Aircraft in correct sector, use double angle intercept */
      /* Find bearing and range to destination and turn shortest direction */
      compute_xz_bearing_dist(current_pos, destination, &true_bearing, &range);
      rel_bearing = relative_bearing(360.0-ry, true_bearing);
      double_heading = normalize_degrees(2.0 * true_bearing - des_heading);
      turn_shortest_heading(double_heading);
      break;

    case 3:
      /* aircraft is in front of desigated heading.  Turn to parallel */
      /* reciprical heading and wait until able to turn inbound  */
      turn_shortest_heading(recip_heading(des_heading));
      compute_xz_bearing_dist(current_pos, destination, &true_bearing, &range);
      rel_bearing = relative_bearing(360.0-ry, true_bearing);
      /* When rel bearing falls enough, start inbound turn */
      if (((rel_bearing > 135.0) && (rel_bearing < 225.0))
          && (range > transition_dist))
        dftp_state_variable = 2;
      break;
    default:
      dftp_state_variable = 0;
      break;
    }  /* end switch */

    /* Signal that we aren't there yet */
    return(0);
  } /* end else */
}  /* end designated_fly_to_point */
```

APPENDIX C. CLIPS SHELL RULES

```
;*******************************************************************
;
; p3_driver.clp
;
; Author:  Dennis A. Schmidt
;         5 April 1993
;
; Comments: Contains clips driving rules for P-3 Expert system
;
;*******************************************************************


;*****************************************************************
;
; Global Variable Definitions
;
;*****************************************************************


(defglobal
  ?*scriptmode* = FALSE              ; Flag for aircraft script
  ?*inbuoycount* = 1           ; count current buoy input
  ?*prosecution* = FALSE
  ?*attackauthorization* = FALSE
  ?*best-buoy-1* = 0
  ?*best-buoy-2* = 0
  ?*sub-posit-x* = 0.0
  ?*sub-posit-z* = 0.0
)

;*****************************************************************
;
; Initialization Rules
;
;*****************************************************************
x
; Start system by reading buoy pattern from file
(defrule init-p3
  ?start-flag <- (initial-fact)
=>
  (retract ?start-flag)
  (init_off_objects)
  (open “waypoint.dat” buoyfile)
  (assert (read-file))
)
```

```
; Reset system if ASW deselected
(defrule reset-problem
  (continue-computing)
  (prosecuting)
  (test (eq ?*prosecution* FALSE))
=>

(printout t "#################### resetting ################" crlf)
(assert (reset-flag))
)

(defrule reset-step2
  (continue-computing)
  (reset-flag)
=>
  (reset)
)

;*******************************************************************
;
;  Rules for executing the main graphics loop
;
;*******************************************************************


; calls P-3 Main driving program
; infinite loop replaces the while(TRUE) graphics loop
(defrule main-p3
    ?start-flag <- (ok-to-draw)
=>
   (retract ?start-flag)          ; retract firing fact
   (p3_main)                      ; call main loop in p3 program
   (assert (ok-to-compute))         ; activate getting script flag
 )

;*******************************************************************
;
;  Rules for reading in initial buoy pattern
;
;*******************************************************************


(defrule read-input-file
  ?read-file <- (read-file)
=>
  (retract ?read-file)
 (assert (data-read =(readline buoyfile))))
```

```
(defrule read-file
  ?data-read <- (data-read ?input&~EOF)      ; check input is not EOF
=>
  (retract ?data-read)
  (printout t ?input crlf)
  (bind ?values (str-explode ?input))
  (bind ?buoy-x (nth 1 ?values))
  (bind ?buoy-z (nth 3 ?values))
  (bind ?buoy-head (nth 4 ?values))
  (assert (drop-buoy ?*inbuoycount* ?buoy-x ?buoy-z ?buoy-head))
  (bind ?*inbuoycount* (+ ?*inbuoycount* 1))
  (assert (read-file)))

(defrule close-all-files
  ?close-files <- (data-read EOF)
=>
  (retract ?close-files)
  (bind ?count ?*inbuoycount*)
  (assert (total-buoys-in-pattern ?count))
  (close)
  (assert (ok-to-compute))
)


;*****************************************************************
;
;  Rules for reading in flags and variables from C and performing calcs
;
;*****************************************************************


; Retrieve current prosecution and script flags (from C)
(defrule get-script-flag
  ?script-var <- (ok-to-compute)
=>

  (retract ?script-var)
  (bind ?*scriptmode* (get_scriptmode))
  (bind ?*prosecution* (get_prosecutionmode))
  (bind ?*attackauthorization* (get_attack_auth))
  (assert (continue-computing))
)

;Rule to process next cycle of scripted sequence
(defrule proc-script
  (continue-computing)
  (test (eq ?*scriptmode* TRUE))
=>
  (process_script)            ; process next step in p3 script
)
```

```
;***************************************************************
;
;  Rules for dropping buoy pattern in sequence
;
;***************************************************************

;Rule to set flag to start drawing rule.  Set to low salience so all
;other computing facts have fired
(defrule end-compute-sequence
  (declare (salience -10))
  ?x <- (continue-computing)
=>
  (retract ?x)
  (assert (ok-to-draw))
)

; Rule to start dropping sonobuoy pattern
(defrule start-prosecution
  (continue-computing)
  ?x <- (not-prosecuting)
  (test (eq ?*prosecution* TRUE))
=>
  (retract ?x)
  (set_speed 240.0)
  (assert (prosecuting))
  (assert (next-buoy 1))
  (assert (start-drops))
)

; Rule to continue dropping sonobuoy pattern
(defrule continue-prosecution
  (continue-computing)
  (test (eq ?*prosecution* TRUE))
=>
  (assert (start-drops))

)

; Increment value identifying next buoy to be dropped
(defrule keep-dropping
  (continue-computing)
  ?cont <- (get-next-buoy)
  ?next <- (next-buoy ?next-num)
=>
  (retract ?cont ?next)
  (assert (next-buoy = (+ ?next-num 1)))
)
```

```
; Check to see if all buoys have been dropped
(defrule is-pattern-complete
 (continue-computing)
 (total-buoys-in-pattern ?count)
 ?next <- (next-buoy ?x&:(= ?x ?count))  ; Is the count equal max number?
=>
 (retract ?next)
 (assert (pattern-complete))
)

; Rule to find the next buoy on the list
(defrule drop-buoys
 (continue-computing)
 (start-drops)
 (next-buoy ?next-num)              ;See which buoy is next
 ?buoy-fact <- (drop-buoy           ; Get posit for that buoy
               ?x&:(= ?x ?next-num)
               ?buoy-x ?buoy-z
               ?buoy-head)
 =>
 (retract ?buoy-fact)
;  (printout t "flying to point "
;        ?x "  " ?buoy-x "  " ?buoy-z "  " ?buoy-head crlf)
 (assert (fly_to =(fly_point ?buoy-x ?buoy-z ?buoy-head)
          ?x ?buoy-x ?buoy-z ?buoy-head))

 )

 ; We haven't reached the fly-to-point, so re-assert fact
(defrule not-reached-buoy-drop-point
  (continue-computing)
  ?y <- (start-drops)
  ?x <- (fly_to FALSE ?bnum ?buoy-x ?buoy-z ?buoy-head)
=>
  (retract ?x ?y)
;   (printout t "continuing flight to point " ?bnum crlf)
  (assert (drop-buoy ?bnum ?buoy-x ?buoy-z ?buoy-head))
)

; We have reached fly-to-point, drop buoy and put continue drop fact out
(defrule reached-buoy-drop-point
  (continue-computing)
  ?y <- (start-drops)
  ?x <- (fly_to TRUE ?bnum ?buoy-x ?buoy-z ?buoy-head)
=>
  (retract ?x ?y)
  (dropbuoy)
  (assert (get-next-buoy))
)
```

```
;*****************************************************************
;
;  Rules to control aircraft after pattern is laid
;
;*****************************************************************


; Pattern is complete orbit until we get contact
(defrule after-pattern-orbit
  (continue-computing)
  (pattern-complete)
  (or (not-attacking) (finished-attacking))
=>
  (fly_point 0.0 -200.0 999.0)
)


; Pattern is complete, monitor buoys for contact
(defrule monitor-pattern
  (continue-computing)
  (pattern-complete)
=>
  (monitor_sonobuoys)
  (bind ?coords (get_best_2_buoys))
  (bind ?*best-buoy-1* (nth 1 ?coords))
  (bind ?*best-buoy-2* (nth 5 ?coords))
  (assert (buoys-updated))

)

; Take best 2 buoys and get a fix
(defrule get-fix
  (continue-computing)
  (pattern-complete)
  ?temp <- (buoys-updated)
=>
  (retract ?temp)
  (bind ?buoy1 ?*best-buoy-1*)
  (bind ?buoy2 ?*best-buoy-2*)
  (bind ?sub-pos (get_sub_fix ?buoy1 ?buoy2))
  (bind ?*sub-posit-x* (nth 1 ?sub-pos))
  (bind ?*sub-posit-z* (nth 2 ?sub-pos))
  (assert (contact))
)
```

```
;*****************************************************************
;
;  Rules for attacking submarine
;
;*****************************************************************

; Rule for attacking submarine
(defrule attack
  (continue-computing)
  (pattern-complete)
  (contact)
  (test (eq ?*attackauthorization* TRUE))
  ?x <- (not-attacking)
=>
  (retract ?x)
  (printout t "Attack authorization received, attacking submarine" crlf)
  (assert (attack-submarine))
)

 ; Start flying to the attack point on shortest heading
(defrule attack1
  (continue-computing)
  (attack-submarine)
  (sub-position ?sub-x ?sub-z)
=>
  (bind ?sub-x ?*sub-posit-x*)
  (bind ?sub-z ?*sub-posit-z*)
  (assert(fly-to-sub =(fly_point ?sub-x ?sub-z 999.0)))
)

 ; We haven't reached the target, so re-assert fact
(defrule not-reached-torp-drop-point
   (continue-computing)
   (attack-submarine)
   ?x <- (fly-to-sub FALSE)
=>
   (retract ?x)
)

; We have reached target, drop torp and signal attack is complete
(defrule reached-torp-drop-point
   (continue-computing)
   ?y <- (attack-submarine)
   ?x <- (fly-to-sub TRUE)
=>
   (retract ?x ?y)
   (droptorp)
   (assert (finished-attacking))
)
```

```
;*****************************************************************
;
; Initial facts on start up
;
;*****************************************************************

(deffacts start
  (initial-fact)
  (not-prosecuting)
  (negative-contact)
  (best-buoys 0 0)
  (sub-position 0.0 0.0)
  (not-attacking)
)
```

LIST OF REFERENCES

[Arvo91] Arvo, James, *Graphics Gems II*, Academic Press, Inc., Boston, 1991.

[Barz92] Barzel, Ronan, *Physically Based Modeling for Computer Graphics, A Structured Approach*, Academic Press, Inc., San Diego, CA, 1992.

[Blau92] Blau, Brian, Hughes, Charles E., Moshell, J. Michael and Lisle, Curtis, "Networked Virtual Environments", *1992 Symposium on Interactive 3D Graphics*, 30 March, 1992, pp. 157-164.

[Bonn88] Bonnet, A., Haton, J. P., Troung-Ngoc, J. M., *Expert Systems, Principles and Practice*, Prentice Hall International (UK) Ltd., London, England, 1988.

[Cook92] Cooke, Joseph M., *NPSNET: Flight Simulation Dynamic Modeling Using Quaternions,* M.S. Thesis, Naval Postgraduate School, Monterey, CA, March 1992.

[Hall88] Halliday, David, Resnick, Robert, *Fundamentals of Physics, Third Edition Extended*, John Wiley & Sons, Inc., New York, NY, 1988.

[Fole87] Foley, James D., van Dam, Andries, Feiner, Steven K., Hughes, John F., *Computer Graphics Principles and Practice,* Addison-Wesley Publishing Company, Reading, MA, 1987.

[Finn90] Finney, Ross L., Thomas, George B. Jr., *Calculus*, Addison-Wesley Publishing Company, Reading, MA, 1990.

[Giar91] Giarratano, Joseph C., *CLIPS User's Guide, Vols 1 & 2,* Software Technology Branch, NASA - Lyndon B. Johnson Space Center, Houston, TX, January 1991.

[IST91] Institute for Simulation and Training, "*Protocol Data Units for Entity Information and Entity Interaction in a Distributed Interactive Simulation*", Military Standard (DRAFT), IST-PD-90-2, Orlando, FL, September 1991.

[Lock] Locke, John, Pratt, David R., and Zyda, Michael J., *A DIS Network Library for UNIX and NPSNET*, Naval Postgraduate School, Monterey, CA, undated.

[NASA91] Software Technology Branch, *CLIPS Reference Manual, Vols I - III,* NASA - Lyndon B. Johnson Space Center, Houston, TX, January 1991.

[NTPS83] Commander Naval Air Systems Command, "NATOPS Flight Manual, Navy Model P-3C Aircraft", NAVAIR 01-75PAC-1, Washington, D. C., 1 December 1983.

[Pope89] Pope, Arthur, “The SIMNET Network and Protocols”, BBN Report No. 7102, BBN Systems and Technologies, Cambridge, MA, July, 1989.

[Rowe88] Rowe, Neil C., *Artificial Intelligence Through Prolog*, Prentice Hall, Inc., EnglewooCliffs, NJ, 1988.

[SGI91] Silicon Graphics, Inc., “Graphics Library Programming Guide”, Document Number 007-1210-040, Mountain View, CA, 1991.

[Ware90] Ware, Colin, and Osborne, Steven, “Exploration and Virtual Camera Control in Virtual Three Dimensional Environments”, *Proceedings, 1990 Symposium on Interactive 3D Graphics, Snowbird, Utah,* 25th - 28th March, 1990, pp. 175-183.

[Watt92] Watt, Alan, Watt, Mark, *Advanced Animation and Rendering Techniques, Theory and Practice*, Addison-Wesley Publishing Company, New York, NY, 1992.

[Weas66] Weast, Robert C., *CRC Handbook of Chemistry and Physics, Forty- Seventh Edition*, The Chemical Rubber Company, Cleveland, OH, 1966.

[Wils92] Wilson, Kalin P., Zyda, Michael J., and Pratt, David R., “NPSGDL: An Object Oriented Graphics Description Language for Virtual Workd Application Support”, *Proceedings of the Third Eurographics Workshop on Object-Oriented Graphics, Champery, Switzerland,* 28-30 October 1992.

[Zyda92] Zyda, Michael J., Pratt, David R., Monahan, James D., and Wilson, Kalin P., “NPSNET: Constructing a 3D Virtual World.”, *1992 Symposium on Interactive 3D Graphics*, 30 March, 1992, pp. 147-156.

[Zyda93] Zyda, Michael J., Wilson, Kalin P., Pratt, David R., Monahan, James G. and Falby, John S. “NPSOFF: An object Description Language for Supporting Virtual World Construction”, *Computers & Graphics,* accepted for Vol. 17, No. 4, 1993.

INITIAL DISTRIBUTION LIST

Defense Technical Information Center 2
Cameron Station
Alexandria, VA 22304-6145

Dudley Knox Library 2
Code 052
Naval Postgraduate School
Monterey, CA 93943-5002

Dr. Michael J. Zyda 2
Computer Science Department Code CS/ZK
Naval Postgraduate School
Monterey, CA 93943

Mr. David R. Pratt 2
Computer Science Department Code CS/PR
Naval Postgraduate School
Monterey, CA 93943

CDR Dennis A. Schmidt 1
Joint Special Operations Command
Code J64
P. O. Box 70239
Fort Bragg, NC 28307-5000